**7**

**Selecting the Right Database Service**

Building applications is all about data collection and management. If you design an e-commerce application, you want to show available inventory catalog data to customers and collect purchase data as they make a transaction. Similarly, if you are running an autonomous vehicle application, you want to analyze data on the surrounding traffic and provide the right prediction to cars based on that data. As of now, you have learned about networking, storage, and compute in previous chapters. In this chapter, you will learn the choices of database services available in AWS to complete the core architecture tech stack.

With so many choices at your disposal, it is easy to get analysis paralysis. So, in this chapter, we will first lay a foundation of how the databases and their use cases can be classified and then use these classifications to help us pick the right service for our particular use case and our circumstances. In this chapter, you will navigate the variety of options, which will give you the confidence that you are using the right tool for the job.

In this chapter, you will learn the following topics:

* A brief history of databases and data-driven innovation trends
* Database consistency model
* Database usages model
* AWS relational and non-relational database services
* Benefits of AWS database services
* Choosing the right tool for the job
* Migrating databases to AWS

By the end of this chapter, you will learn about different AWS database service offerings and how to choose a suitable database for your workload.

**A brief history of databases**

Relational databases have been around for over 50 years. Edgar F. Codd created the first database in 1970. The main feature of a relational database is that data is arranged in rows and columns, and rows in tables are associated with other rows in other tables by using the column values in each row as relationship keys. Another important feature of relational databases is that they normally use **Structured Query Language** (**SQL**) to access, insert, update, and delete records. SQL was created by IBM researchers Raymond Boyce and Donald Chamberlin in the 1970s. Relational databases and SQL have served us well for decades.

As the internet’s popularity increased in the 1990s, we started hitting scalability limits with relational databases. Additionally, a wider variety of data types started cropping up. **Relational Database Management Systems** (**RDBMSs**) were simply not enough anymore. This led to the development of new designs, and we got the term **NoSQL databases**. As confusing as the term is, it does convey the idea that it can deal with data that is not structured, and it deals with it with more flexibility.

The term **NoSQL** is attributed to Carlo Strozzi and was first used in 1998 for a relational database that he developed but that didn’t use the SQL language. The term was then again used in 2009 by Eric Evans and Johan Oskarsson to describe databases that were not relational.

The main difference between relational and non-relational databases is the way they store data and query it. Let’s see an example of making a choice between a relational and non-relational database. Take an example of a banking transaction; it is critical for financial transactions in every customer’s bank account to always be consistent and roll back in case of any error. In such a scenario, you want to use a relational database. For a relational database, if some information is not available, then you are forced to store null or some other value. Now take an example of a social media profile, which may have hundreds of attributes to store the user’s name, address, education, jobs, personal choices, preferences, etc. However, many users do not fill in all the information; some users may add just their name, while others add more details like their address and education. In such cases, you want to use a non-relational database and store only the information provided by the user without adding null values where the user doesn’t provide details (unlike in relational databases).

The following tables demonstrate the difference between relational and non-relational databases that have the same user data.

|  |  |  |  |
| --- | --- | --- | --- |
| **First Name** | **Last Name** | **City** | **Country** |
| **Maverick** | Doe | Seattle | USA |
| **Goose** | Henske | NULL | NULL |
| **John** | Gayle | London | NULL |

*Table 7.1: Relational database*

|  |  |  |  |
| --- | --- | --- | --- |
| **First Name** | **Last Name** | **City** | **Country** |
| **Maverick** | Doe | Seattle | USA |
| **Goose** | Henske |  |  |
| **John** | Gayle | London |  |

*Table 7.2: Non-relational database*

In the tables above, 3 users provided their information with their first name, last name, city, and country. You can see that only Maverick provided their full information while the other users left out either their city, country, or both. In a relational database, you need to fill in missing information with a NULL value across all columns, while in a non-relational database, that column doesn’t exist at all.

It is nothing short of amazing what has occurred since then. Hundreds of new offerings have been developed, each trying to solve a different problem. In this environment, deciding the best service or product to solve your problem becomes complicated. And you must consider not only your current requirements and workloads, but also take into account that your choice of database will be able to cover your future requirements and new demands. With so much data getting generated, it is natural that much of innovation is driven by data. Let’s look in detail at how data is driving innovation.

**Data-driven innovation trends**

Since high-speed internet became available in the last decade, more and more data is getting generated. Before we proceed, let’s discuss three significant trends that influence your perspective on data:

* **The surge of data**: Our current era is witnessing an enormous surge in data generation. Managing the vast amount of data originating from your business applications is essential.

However, the exponential growth primarily stems from the data produced by network-connected intelligent devices, amplifying the data’s diversity and quantity. These “smart” devices, including but not limited to mobile phones, connected vehicles, smart homes, wearable technologies, household appliances, security systems, industrial equipment, machinery, and electronic gadgets, constantly generate real-time data. Notably, over one-third of mobile sign-ups on cellular networks result from built-in cellular connections in most modern cars. In addition, applications generate real-time data, such as purchase data from e-commerce sites, user behavior from mobile apps, and social media posts/tweets. The data volume is expanding tenfold every five years, necessitating cloud-based solutions to manage and exploit vast data efficiently.

* **Microservices change analytics requirements**: The advent of microservices is revolutionizing organizations’ data and analytics requirements. Rather than developing monolithic applications, companies are shifting towards a microservices architecture that divides complex problems into independent units. This approach enables developers to operate in smaller groups with minimal coordination, respond more efficiently, and work faster. Microservices enable developers to break down their applications into smaller parts, providing them with the flexibility to use multiple databases for various workloads, each suited for its specific purpose. The importance of analytics cannot be overstated, and it must be incorporated into every aspect of the business, rather than just being an after-the-fact activity. Monitoring the organization’s operations in real time is critical to fuel innovation and quick decision-making, whether through human intervention or automated processes. Today’s well-run businesses thrive on the swift utilization of data.
* **DevOps driving fast changes**: The fast-paced rate of change, driven by DevOps, is transforming how businesses approach IT. To keep up with the rapid innovation and the velocity of IT changes, organizations are adopting the DevOps model. This approach employs automated development tools to facilitate continuous software development, deployment, and enhancement. DevOps emphasizes effective communication, collaboration, and integration between software developers and IT operations. It also involves a rapid rate of change and change management, enabling businesses to adapt to evolving market needs and stay ahead of the competition.

While you see the trend that the industry is adopting, let’s learn some basics of databases and learn about the database consistency model in more detail.

**Database consistency model**

In the context of databases, ensuring transaction data consistency involves restricting any database transaction’s ability to modify data in unauthorized ways. When data is written to the database, it must adhere to a set of predefined rules and constraints. These rules are verified, and all checks must be successfully passed before the data can be accessed by other users. This stringent process ensures that data integrity is maintained and that the information stored in the database is accurate and trustworthy. Currently, there are two popular data consistency models. We’ll discuss these models in the following subsections.

**ACID data consistency model**

When database sizes were measured in megabytes, we could have stringent requirements that enforced strict consistency. Since storage has become exponentially cheaper, databases can be much bigger, often measured in terabytes and even petabytes. For this reason, making databases ACID-compliant for storage reasons is much less prevalent. The ACID model guarantees the following:

* **Atomicity**: For an operation to be considered atomic, it should ensure that transactions within the operation either succeed or fail. If one of the transactions fails, all operations should fail and be rolled back. Could you imagine what would happen if you went to the ATM and the machine gave you money but didn’t deduct it from your account?
* **Consistency**: The database is structurally sound and consistent after completing each transaction.
* **Isolation**: Transactions are isolated and don’t contend with each other. Access to data from multiple users is moderated to avoid contention. Isolation guarantees that two transactions cannot coincide.
* **Durability**: After a transaction is completed, any changes a transaction makes should be durable and permanent, even in a failure such as a power failure.

The ACID model came before the BASE model, which we will describe next. If performance were not a consideration, using the ACID model would always be the right choice. BASE only came into the picture because the ACID model could not scale in many instances, especially with internet applications that serve a worldwide client base.

**BASE data consistency model**

ACID was taken as the law of the land for many years, but a new model emerged with the advent of bigger-scale projects and implementations. In many instances, the ACID model is more pessimistic than required, and it’s *too safe* at the expense of scalability and performance.

In most NoSQL databases, the ACID model is not used. These databases have loosened some ACID requirements, such as data freshness, immediate consistency, and accuracy, to gain other benefits, such as scale, speed, and resilience. Some exceptions for a NoSQL database that uses the ACID models are the NET-based RavenDB database and Amazon DynamoDB within a single AWS account and region.

The acronym **BASE** can be broken down as follows – **Basic Availability**, **Soft-state**, and **Eventual consistency**. Let’s explore what this means further:

* **Basic availability**: The data is available for the majority of the time (but not necessarily all the time). The BASE model emphasizes availability without guaranteeing the consistency of data replication when writing a record.
* **Soft-state**: The database doesn’t have to be write-consistent, and different replicas don’t always have to be mutually consistent. Take, for example, a system that reports sales figures in real-time to multiple destinations and uses multiple copies of the sales figures to provide fault tolerance. As sales come in and get written into the system, different readers may read a different copy of the sales figures. Some of them may be updated with the new numbers, and others may be a few milliseconds behind and not have the latest updates. In this case, the readers will have different results, but if they rerun the query soon after, they probably would get the new figures. In a system like this, not having the latest and greatest numbers may not end the world and may be good enough. The trade-off between getting the results fast versus being entirely up to date may be acceptable.
* **Eventual consistency**: The stored data exhibits consistency eventually and maybe not until the data is retrieved at a later point.

The BASE model requirements are looser than the ACID model ones, and a direct one-for-one relationship does not exist between ACID and BASE. The BASE consistency model is used mainly in aggregate databases (including wide-column databases), key-value databases, and document databases.

Let’s look at the database usage model, which is a crucial differentiator when storing your data.

**Database usage model**

Two operations can be performed with a database: first, ingest data (or write data into the database), and second, retrieve data (or read data from the database). These two operations will always be present.

On the ingestion side, the data will be ingested in two different ways. It will either be a data update or brand-new data (such as an insert operation). To retrieve data, you will analyze the **change data capture** (**CDC**) set, which is changes in existing data or accessing brand new data. But what drives your choice of database is not the fact that these two operations are present but rather the following:

* How often will the data be retrieved?
* How fast should it be accessed?
* Will the data be updated often, or will it be primarily new?
* How often will the data be ingested?
* How fast does ingestion need to be?
* Will the ingested data be sent in batches or in real time?
* How many users will be consuming the data?
* How many simultaneous processes will there be for ingestion?

The answers to these questions will determine what database technology to use. Two technologies have been the standards to address these questions for many years: **online transaction processing** (**OLTP**) systems and **online analytics processing** (**OLAP**) systems. The main question that needs to be answered is - *is it more important for the database to perform during data ingestion or retrieval?* These databases can be divided into two categories depending on the use case; they need to be read-heavy or write-heavy.

**Online Transaction Processing (OLTP) systems**

OLTP databases’ main characteristics are the fact that they process a large number of transactions (such as inserts and updates). The focus in OLTP systems is placed on fast ingestion and modification of data while maintaining data integrity, typically in a multi-user environment with less emphasis on the retrieval of the data. OLTP performance is generally measured by the number of transactions executed in a given time (usually seconds). Data is typically stored using a schema that has been normalized, usually using the **3rd normal form** (**3NF**). Before moving on, let’s quickly discuss 3NF. 3NF is a state that a relational database schema design can possess.

A table using 3NF will reduce data duplication, minimize data anomalies, guarantee referential integrity, and increase data management. 3NF was first specified in 1971 by *Edgar F. Codd*, the inventor of the relational model for database management.

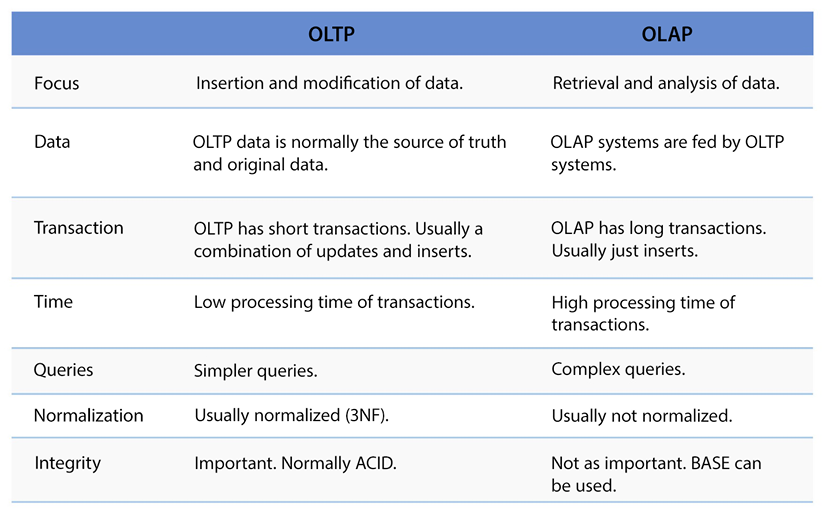
A database relation (for example, a database table) meets the 3NF standard if each table’s columns only depend on the table’s primary key. Let’s look at an example of a table that fails to meet 3NF. Let’s say you have a table that contains a list of employees. This table, in addition to other columns, contains the employee’s supervisor’s name as well as the supervisor’s phone number. A supervisor can undoubtedly have more than one employee under supervision, so the supervisor’s name and phone number will be repeated for employees working under the same supervisor. To resolve this issue, we could add a supervisor table, put the supervisor’s name and phone number in the supervisor table, and remove the phone number from the employee table.

**Online Analytical Processing (OLAP) systems**

Conversely, OLAP databases do not process many transactions. Once data is ingested, it is usually not modified. OLTP systems are not uncommon to be the source systems for OLAP systems. Data retrieval is often performed using some query language (the **Structured Query Language** (**SQL**)). Queries in an OLAP environment are often complex and involve subqueries and aggregations. In the context of OLAP systems, the performance of queries is the relevant measure. An OLAP database typically contains historical data aggregated and stored in multi-dimensional schemas (typically using the star schema).

For example, a bank might handle millions of daily transactions, storing deposits, withdrawals, and other banking data. The initial transactions will probably be stored using an OLTP system. The data might be copied to an OLAP system to run reporting based on the daily transactions and, once aggregated, for more extended reporting periods.

The following table shows a comparison between OLTP and OLAP:



*Figure 7.1: Comparison between OLTP systems and OLAP systems*

As you have learned about the database consistency model and its uses, you must be wondering which model is suitable when combining these properties; ACID is a must-have for OLTP, and BASE can be applied for OLAP.

Let’s go further and learn about the various kinds of database services available in AWS and how they fit to address different workload needs.

**AWS database services**

AWS offers a broad range of database services that are purpose-built for every major use case. These fully managed services allow you to build applications that scale quickly. All these services are battle-tested and provide deep functionality, so you get the high availability, performance, reliability, and security required by production workloads.

The suite of AWS fully managed database services encompasses relational databases for transactional applications, such as Amazon RDS and Amazon Aurora, non-relational databases like Amazon DynamoDB for internet-scale applications, an in-memory data store called Amazon ElastiCache for caching and real-time workloads, and a graph database, Amazon Neptune, for developing applications with highly connected data. Migrating your existing databases to AWS is made simple and cost-effective with the AWS Database Migration Service. Each of these database services is so vast that going into details warrants a book for each of these services itself. This section will show you various database services overviews and resources to dive further.

**Relational databases**

There are many offerings in the database space, but relational databases have served us well for many years without needing any other type of database. A relational database is probably the best, cheapest, and most efficient option for any project that does not store millions of records. So, let’s analyze the different relational options that AWS offers us.

**Amazon Relational Database Service (Amazon RDS)**

Given what we said in the previous section, it is not surprising that Amazon has a robust lineup of relational database offerings. They all fall under the umbrella of Amazon RDS. It is certainly possible to install your database into an EC2 instance and manage it yourself. Unless you have an excellent reason to do so, it may be a terrible idea; instead, you should consider using one of the many flavors of Amazon RDS. You may think running your instance might be cheaper, but if you consider all the costs, including system administration costs, you will most likely be better off and save money using Amazon RDS.

Amazon RDS was designed by AWS to simplify the management of crucial transactional applications by providing an easy-to-use platform for setting up, operating, and scaling a relational database in the cloud. With RDS, laborious administrative tasks such as hardware provisioning, database configuration, patching, and backups are automated, and a scalable capacity is provided in a cost-efficient manner. RDS is available on various database instance types, optimized for memory, performance, or I/O, and supports six well-known database engines, including Amazon Aurora (compatible with MySQL and PostgreSQL), MySQL, PostgreSQL, MariaDB, SQL Server, and Oracle.

If you want more control of your database at the OS level, AWS has now launched **Amazon RDS Custom**. It provisions all AWS resources in your account, enabling full access to the underlying Amazon EC2 resources and database environment access.

You can install third-party and packaged applications directly onto the database instance as they would have in a self-managed environment while benefiting from the automation that Amazon RDS traditionally provides.

Amazon RDS’s flavors fall into three broad categories:

* **Community** (Postgres, MySQL, and MariaDB): AWS offers RDS with three different open-source offerings. This is a good option for development environments, low-usage deployments, defined workloads, and non-critical applications that can afford some downtime.
* **Amazon Aurora** (Postgres and MySQL): As you can see, Postgres and MySQL are here, as they are in the community editions. Is this a typo? No, delivering these applications within the Aurora *wrapper* can add many benefits to a community deployment. Amazon started offering the MySQL service in 2014 and added the Postgres version in 2017. Some of these are as follows:
  1. Automatic allocation of storage space in 10 GB increments up to 64 TBs
  2. Fivefold performance increase over the vanilla MySQL version
  3. Automatic six-way replication across availability zones to improve availability and fault tolerance
* **Commercial** (Oracle and SQLServer): Many organizations still run Oracle workloads, so AWS offer RDS with an Oracle flavor (and a Microsoft SQL Server flavor). Here, you will get all the benefits of a fully managed service. However, keep in mind that, bundled with the cost of this service, there will be a licensing cost associated with using this service, which otherwise might not be present if you use a community edition.

Let’s look at the key benefits of Amazon RDS.

**Amazon RDS Benefits**

Amazon RDS offers multiple benefits as a managed database service offered by AWS. Let’s look at its key attributes to make your database more resilient and performant.

**Multi-AZ deployments** - Multi-AZ deployments in RDS provide improved availability and durability for database instances, making them an ideal choice for production database workloads. With Multi-AZ DB instances, RDS synchronously replicates data to a standby instance in a different **Availability Zone** (**AZ**) for enhanced resilience. You can change your environment from Single-AZ to Multi-AZ at any time. Each AZ runs on its own distinct, independent infrastructure and is built to be highly dependable.

In the event of an infrastructure failure, RDS initiates an automatic failover to the standby instance, allowing you to resume database operations as soon as the failover is complete. Additionally, the endpoint for your DB instance remains the same after a failover, eliminating manual administrative intervention and enabling your application to resume database operations seamlessly.

**Read replicas** - RDS makes it easy to create read replicas of your database and automatically keeps them in sync with the primary database (for MySQL, PostgreSQL, and MariaDB engines). Read replicas are helpful for both read scaling and disaster recovery use cases. You can add read replicas to handle read workloads, so your master database doesn’t become overloaded with reading requests. Depending on the database engine, you may be able to position your read replica in a different region than your master, providing you with the option of having a read location that is closer to a specific locality. Furthermore, read replicas provide an additional option for failover in case of an issue with the master, ensuring you have coverage in the event of a disaster.

While both Multi-AZ deployments and read replicas can be used independently, they can also be used together to provide even greater availability and performance for your database. In this case, you would create a Multi-AZ deployment for your primary database and then create one or more read replicas of that primary database. This would allow you to benefit from the automatic failover capabilities of Multi-AZ deployments and the performance improvements provided by read replicas.

**Automated backup** - With RDS, a scheduled backup is automatically performed once a day during a time window that you can specify. The backup job is monitored as a managed service to ensure its successful completion within the specified time window. The backups are comprehensive and include both your entire instance and transaction logs. You have the flexibility to choose the retention period for your backups, which can be up to 35 days. While automated backups are available for 35 days, you can retain longer backups using the manual snapshots feature provided by RDS. RDS keeps multiple copies of your backup in each AZ where you have an instance deployed to ensure their durability and availability. During the automatic backup window, storage I/O might be briefly suspended while the backup process initializes, typically for less than a few seconds. This may cause a brief period of elevated latency. However, no I/O suspension occurs for Multi-AZ DB deployments because the backup is taken from the standby instance. This can help achieve high performance if your application is time-sensitive and needs to be always on.

**Database snapshots** - You can manually create backups of your instance stored in Amazon S3, which are retained until you decide to remove them. You can use a database snapshot to create a new instance whenever needed. Even though database snapshots function as complete backups, you are charged only for incremental storage usage.

**Data storage** - Amazon RDS supports the most demanding database applications by utilizing **Amazon Elastic Block Store** (**Amazon EBS**) volumes for database and log storage. There are two SSD-backed storage options to choose from: a cost-effective general-purpose option and a high-performance OLTP option. Amazon RDS automatically stripes across multiple Amazon EBS volumes to improve performance based on the requested storage amount.

**Scalability** - You can often scale your RDS database compute and storage resources without downtime. You can choose from over 25 instance types to find the best fit for your CPU, memory, and price requirements. You may want to scale your database instance up or down, including scaling up to handle the higher load, scaling down to preserve resources when you have a lower load, and scaling up and down to control costs if you have regular periods of high and low usage.

**Monitoring** - RDS offers a set of 15-18 monitoring metrics that are automatically available for you. You can access these metrics through the RDS or CloudWatch APIs. These metrics enable you to monitor crucial aspects such as CPU utilization, memory usage, storage, and latency. You can view the metrics in individual or multiple graphs or integrate them into your existing monitoring tool. Additionally, RDS provides Enhanced Monitoring, which offers access to more than 50 additional metrics. By enabling Enhanced Monitoring, you can specify the granularity at which you want to view the metrics, ranging from one-second to sixty-second intervals. This feature is available for all six database engines supported by RDS.

**Amazon RDS Performance Insights** is a performance monitoring tool for Amazon RDS databases. It allows you to monitor the performance of your databases in real-time and provides insights and recommendations for improving the performance of your applications. With Performance Insights, you can view a graphical representation of your database’s performance over time and detailed performance metrics for specific database operations. This can help you identify any potential performance bottlenecks or issues and take action to resolve them.

Performance Insights also provides recommendations for improving the performance of your database. These recommendations are based on best practices and the performance data collected by the tool and can help you optimize your database configuration and application code to improve the overall performance of your application.

**Security** - Controlling network access to your database is made simple with RDS. You can run your database instances in **Amazon Virtual Private Cloud** (**Amazon VPC**) to isolate them and establish an industry-standard encrypted IPsec VPN to connect with your existing IT infrastructure. Additionally, most RDS engine types offer encryption at rest, and all engines support encryption in transit. RDS offers a wide range of compliance readiness, including HIPAA eligibility.

You can learn more about RDS by visiting the AWS page: <https://aws.amazon.com/rds/>.

As you have learned about RDS, let’s dive deeper into AWS cloud-native databases with Amazon Aurora.

**Amazon Aurora**

Amazon Aurora is a relational database service that blends the availability and rapidity of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. Aurora is built with full compatibility with MySQL and PostgreSQL engines, enabling applications and tools to operate without necessitating modifications. It offers a variety of developer tools to construct serverless and **machine learning** (**ML**)-driven applications. The service is completely managed and automates time-intensive administration tasks, including hardware provisioning, database setup, patching, and backups. It provides commercial-grade databases’ reliability, availability, and security while costing only a fraction of the price.

Amazon Aurora has many key features that have been added to expand the service’s capabilities since it launched in 2014. Let’s review some of these key features:

* **Serverless configuration** - Amazon Aurora Serverless is a configuration of Aurora that offers auto-scaling features on-demand. With this configuration, your database will automatically start up, shut down, and adjust its capacity based on the needs of your application. Amazon Aurora Serverless v2 scales almost instantly to accommodate hundreds of thousands of transactions in seconds. It fine-tunes its capacity in small increments to ensure the right resources for your application. You won’t have to manage the database capacity, and you’ll only pay for your application’s resources. Compared to peak load provisioning capacity, you could save up to 90% of your database cost with Amazon Aurora Serverless.
* **Global Database** - To support globally distributed applications, you can leverage the Global Database feature of Aurora. This enables you to span a single Aurora database across multiple AWS regions, allowing for faster local reads and rapid disaster recovery. Global Database utilizes storage-based replication to replicate your database across various regions, typically resulting in less than one-second latency. By utilizing a secondary region, you can have a backup option in case of a regional outage or degradation and can quickly recover. Additionally, it takes less than one minute to promote a database in the secondary region to full read/write capabilities.
* **Encryption** - With Amazon Aurora, you can encrypt your databases by using keys you create and manage through **AWS Key Management Service** (**AWS KMS**). When you use Amazon Aurora encryption, data stored on the underlying storage and automated backups, snapshots, and replicas within the same cluster are encrypted. Amazon Aurora secures data in transit using SSL (AES-256).
* **Automatic, continuous, incremental backups and point-in-time restore** - Amazon Aurora offers a backup feature that enables you to recover your instance to any specific second during your retention period, up to the last five minutes. This capability is known as point-in-time recovery. You can configure your automatic backup retention period for up to thirty-five days. The automated backups are stored in **Amazon Simple Storage Service** (**Amazon S3**), which is designed for 99.999999999% durability. The backups are incremental, continuous, and automatic, and they have no impact on database performance.
* **Multi-AZ Deployments with Aurora Replicas** - In the event of an instance failure, Amazon Aurora leverages Amazon RDS Multi-AZ technology to perform an automated failover to one of the up to 15 Amazon Aurora Replicas you have established across three AZs. If you have not provisioned any Amazon Aurora Replicas, in the event of a failure, Amazon RDS will automatically attempt to create a new Amazon Aurora DB instance for you.
* **Compute Scaling** - You can scale the provisioned instances powering your deployment up or down using either the Amazon RDS APIs or the AWS Management Console. The process of compute scaling typically takes only a few minutes to complete.
* **Storage auto-scaling** - Amazon Aurora automatically scales the size of your database volume to accommodate increasing storage requirements. The volume expands in 10 GB increments, up to 128 TB, as your storage needs grow. There’s no need to provision extra storage to handle the future growth of your database.
* **Fault-tolerant and self-healing storage** - Each 10 GB chunk of your database volume is replicated six times across three Availability Zones, making Amazon Aurora storage fault-tolerant. It can handle the loss of up to two data copies without affecting write availability, and up to three copies without affecting read availability. Amazon Aurora storage is also self-healing, continuously scanning data blocks and disks for errors and replacing them automatically.
* **Network isolation** - Amazon Aurora operates within Amazon VPC, providing you with the ability to segregate your database within your own virtual network and connect to your existing on-premises IT infrastructure via industry-standard encrypted IPsec VPNs. Additionally, you can manage firewall configurations through Amazon RDS and govern network access to your DB instances.
* **Monitoring and metrics** - Amazon Aurora offers a range of monitoring and performance tools to help you keep your database instances running smoothly. You can use Amazon CloudWatch metrics at no additional cost to monitor over 20 key operational metrics, such as compute, memory, storage, query throughput, cache hit ratio, and active connections. If you need more detailed insights, you can use Enhanced Monitoring to gather metrics from the operating system instance that your database runs on. Additionally, you can use Amazon RDS Performance Insights, a powerful database monitoring tool that provides an easy-to-understand dashboard for visualizing database load and detecting performance problems, so you can take corrective action quickly.
* **Governance** - AWS CloudTrail keeps track of and documents account activity across your AWS infrastructure, providing you with oversight over storage, analysis, and corrective actions. You can ensure your organization remains compliant with regulations such as SOC, PCI, and HIPAA by utilizing CloudTrail logs. The platform enables you to capture and unify user activity and API usage across AWS Regions and accounts in a centralized, controlled environment, which can help you avoid penalties.
* **Amazon Aurora machine learning** - With Amazon Aurora machine learning, you can incorporate machine learning predictions into your applications through SQL programming language, eliminating the need to acquire separate tools or possess prior machine learning experience. It offers a straightforward, optimized, and secure integration between Aurora and AWS ML services, eliminating the need to create custom integrations or move data between them.

Enterprise use cases for Amazon Aurora span multiple industries. Here are examples of some of the key use cases where Amazon Aurora is an excellent fit, along with specific customer references for each:

* **Revamp corporate applications** - Ensure high availability and performance of enterprise applications, including CRM, ERP, supply chain, and billing applications.
* **Build a Software-as-a-Service (SaaS) application** - Ensure flexible instance and storage scaling to support dependable, high-performing, and multi-tenant SaaS applications. Amazon Aurora is a good choice for building a SaaS application, as it provides the scalability, performance, availability, and security features that are essential for successful SaaS applications. Amazon Aurora automatically creates and maintains multiple replicas of your data, providing high availability and failover capabilities. This ensures that your SaaS application is always available, even in the event of an outage or failure.

Amazon Aurora provides several security features, such as encryption at rest and in transit, to help protect your data and ensure compliance with industry standards and regulations.

* **Deploy globally distributed applications** - Achieve multi-region scalability and resilience for internet-scale applications, such as mobile games, social media apps, and online services, with Aurora’s flexible instance and storage scaling. To meet high read or write requirements, databases are often split across multiple instances, but this can lead to over-provisioning or under-provisioning, resulting in increased costs or limited scalability. Aurora Serverless solves this problem by automatically scaling the capacity of multiple Aurora instances based on the application’s needs, allowing for efficient scaling of databases and enabling the deployment of globally distributed applications. This can provide several benefits, including cost savings, flexibility, and simplicity.
* **Variable and unpredictable workloads** - If you run an infrequently-used application where you need to provision for peak, it will require you to pay for unused resources. A surge in traffic can also be mitigated through the automatic scaling of Aurora Serverless. You will not need to manage or upsize your servers manually. You need to set a min/max capacity unit setting and allow Aurora to scale to meet the load.

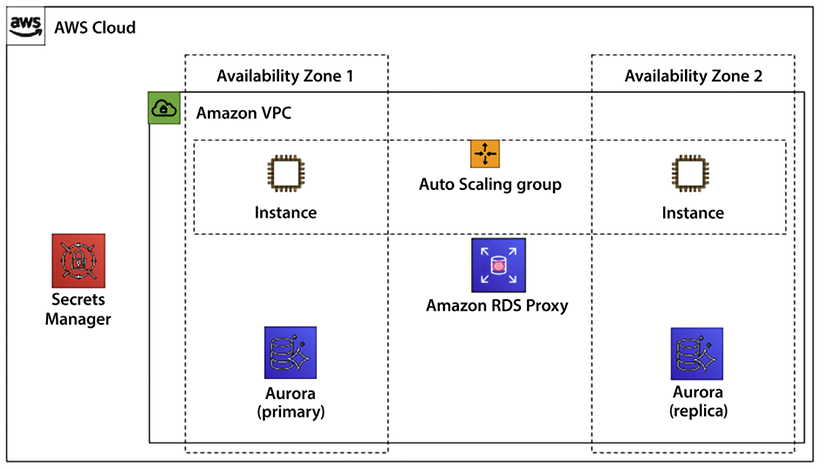
Amazon RDS Proxy works together with Aurora to enhance database efficiency and application scalability by enabling applications to share and pool connections established with the database. Let’s learn more details about RDS Proxy.

**Amazon RDS Proxy**

Amazon RDS Proxy is a service that acts as a database proxy for Amazon **Relational Database Service** (**RDS**). It is fully managed by AWS and helps to increase the scalability and resilience of applications in the face of database failures, while enhancing the security of database traffic. RDS Proxy sits between your application and your database and automatically routes database traffic to the appropriate RDS instances. This can provide several benefits, including:

* **Improved scalability**: RDS Proxy automatically scales to handle a large number of concurrent connections, making it easier for your application to scale.
* **Better resilience to database failures**: RDS Proxy can automatically failover to a standby replica if the primary database instance becomes unavailable, reducing downtime and improving availability.
* **Enhanced security**: RDS Proxy can authenticate and authorize incoming connections, helping to prevent unauthorized access to your database. It can also encrypt data in transit, providing an extra security measure for your data.

The following diagram shows Amazon EC2 web server provision with an Aurora database where database passwords are managed by AWS Secrets Manager. Aurora put across 2 **Availability Zones** (**AZs**) to achieve high availability, where AZ1 hosts Aurora’s primary database while AZ2 has the Aurora read replica.



*Figure 7.2: Amazon Aurora’s high availability with RDS proxy*

With RDS Proxy, when a failover happens, application connections are preserved. Only transactions that are actively sending or processing data will be impacted. During failover, the proxy continues to accept new connections. These connections will queue until the database connection comes online; at that point, it then gets sent over to the database.

Amazon RDS Proxy is a useful tool for improving the performance, availability, and security of your database-powered applications. It is fully managed, so you don’t have to worry about the underlying infrastructure, and it can help make your applications more scalable, resilient, and secure. You can learn more about RDS Proxy by visiting the AWS page: <https://aws.amazon.com/rds/proxy/>.

High availability and performance are a database’s most essential and tricky parts. But this problem can be solved in an intelligent way using machine learning. Let’s look at RDS’s newly launched feature, Amazon DevOps Guru, to help with database performance issues using ML.

**Amazon DevOps Guru for RDS**

DevOps Guru for Amazon RDS is a recently introduced capability that uses machine learning to automatically identify and troubleshoot performance and operational problems related to relational databases in an application. The tool can detect issues such as over-utilization of resources or problematic SQL queries and provide recommendations for resolution, helping developers address these issues quickly. DevOps Guru for Amazon RDS utilizes machine learning models to deliver these insights and suggestions.

DevOps Guru for RDS aids in quickly resolving operational problems related to databases by notifying developers immediately via Amazon **Simple Notification Service** (**SNS**) notifications and EventBridge when issues arise. It also provides diagnostic information, as well as intelligent remediation recommendations, and details on the extent of the issue.

AWS keeps adding innovations for Amazon RDS as a core service. Recently, they launched Amazon RDS instances available on AWS’s chip Graviton2, which helps them offer lower prices with increased performance. RDS is now available in Amazon Outpost to fulfill your need to keep the database near your workload in an on-premise environment. You can learn more about RDS Proxy by visiting the AWS page: <https://aws.amazon.com/devops-guru/>.

Besides SQL databases, Amazon’s well-known NoSQL databases are very popular. Let’s learn about some NoSQL databases.

**AWS NoSQL databases**

When it comes to NoSQL databases, you need to understand two main categories, key-value and document, as those can be confusing. The key-value database needs high throughput, low latency, reads and writes, and endless scale, while the document database stores documents and quickly accesses querying on any attribute. Document and key-value databases are close cousins. Both types of databases rely heavily on keys that point to a value. The main difference between them is that in a document database, the value stored (the document) will be **transparent** to the database and, therefore, can be indexed to assist retrieval. In the case of a key-value database, the value is **opaque** and will not be scannable, indexed, or visible until the value is retrieved by specifying the key. Retrieving a value without using the key would require a full table scan. Content is stored as the value. To retrieve the content, you query using the unique key, enabling access to the value.

Key-value databases are the simpletons of the NoSQL world. They are pretty easy to use. There are three simple API operations:

* Retrieve the value for a key.
* Insert or update a value using a key reference.
* Delete a value using the key reference.

The values are generally **Binary Large Objects** (**BLOBs**). Data stores keep the value without regard for the content. Data in key-value database records is accessed using the key (in rare instances, it might be accessed using other filters or a full table scan). Therefore, performance is high and scalable. The biggest strength of key-value databases is always their biggest weakness. Data access is quick because we use the key to access the data. Still, full table scans and filtering operations are neither supported nor a secondary consideration. Key-value stores often use the hash table pattern to store the keys. No column-type relationships exist, which keeps the implementation details simple. Starting in the key-value category, AWS provides Amazon DynamoDB. Let’s learn more about Dynamo DB.

**Amazon Dynamo DB**

DynamoDB is a fully managed, multi-Region, multi-active database that delivers exceptional performance, with single-digit-millisecond latency, at any scale. It is capable of handling more than 10 trillion daily requests, with the ability to support peaks of over 20 million requests per second, making it an ideal choice for internet-scale applications. DynamoDB offers built-in security, backup and restore features, and in-memory caching. One of the unique features of DynamoDB is its elastic scaling, which allows for seamless growth as the number of users and required I/O throughput increases. You pay only for the storage and I/O throughput you provision, or on a consumption-based model if you choose on-demand. The database can be provisioned with additional capacity on the fly to maintain high performance, making it easy for an application to support millions of users making thousands of concurrent requests every second. In addition, DynamoDB offers fine-grained access control and support for end-to-end encryption to ensure data security.

Some of DynamoDB’s benefits are as follows:

* Fully managed
* Supports multi-region deployment
* Multi-master deployment
* Fine-grained identity and access control
* Seamless integration with IAM security
* In-memory caching for fast retrieval
* Supports ACID transactions
* Encrypts all data by default

DynamoDB provides the option of on-demand backups for archiving data to meet regulatory requirements. This feature enables you to create full backups of your DynamoDB table’s data. Additionally, you can enable continuous backups for point-in-time recovery, allowing restoration to any point in the last 35 days with per-second granularity. All backups are automatically encrypted, cataloged, and retained until explicitly deleted. This feature ensures backups are easily discoverable and helps meet regulatory requirements.

DynamoDB is built for high availability and durability. All writes are persisted on an SSD disk and replicated to 3 availability zones. Reads can be configured as “strong” or “eventual.” There is no latency trade-off with either configuration; however, the read capacity is used differently. Amazon **DynamoDB Accelerator** (**DAX**) is a managed, highly available, in-memory cache for DynamoDB that offers significant performance improvements, up to 10 times faster than standard DynamoDB, even at high request rates. DAX eliminates the need for you to manage cache invalidation, data population, or cluster management, and delivers microseconds of latency by doing all the heavy lifting required to add in-memory acceleration to your DynamoDB tables.

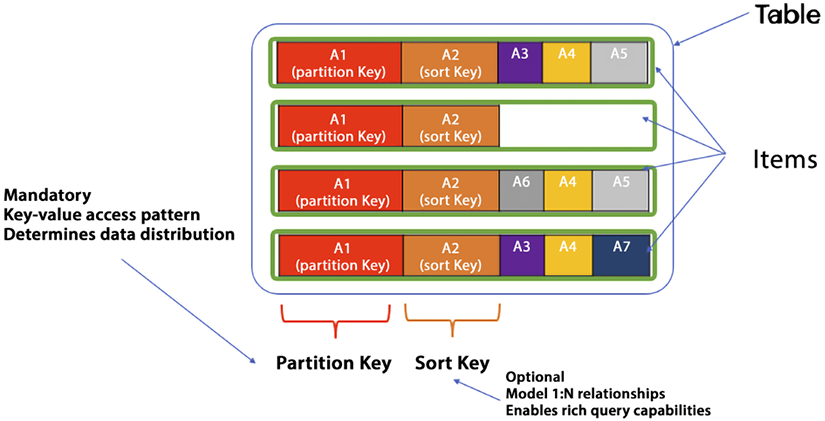
**Defining Dynamo DB Table**

DynamoDB is a table-based database. While creating the table, you can specify at least three components:

1. **Keys**: There will be two parts of the key – first, a partition key to retrieve the data, and second, a sort key to sort and retrieve a batch of data in a given range. For example, transaction ID can be your primary key, and transaction date-time can be the sort key.
2. **WCU**: **Write capacity unit** (1 KB/sec) defines at what rate you want to write your data in DynamoDB.
3. **RCU**: **Read capacity unit** (4 KB/sec) defines at what rate you want to read from your given DynamoDB table.

The size of the table automatically increases as you add more items. There is a hard limit of item size at 400 KB. As size increases, the table is partitioned automatically for you. The size and provisioning capacity of the table are equally distributed for all partitions.

As shown in the below diagram, the data is stored in tables; you can think of a table as the database, and within the table, you have items.



*Figure 7.3: DynamoDB table with partition and sort key*

As shown in the preceding diagram, the first item has five attributes, and the next item has only two. As more items are added to the table in DynamoDB, it becomes apparent that attributes can differ between items, and each item can have a unique set of attributes. Additionally, the primary key or partition key can be observed, which uniquely identifies each item and determines how the data is partitioned and stored. The partition key is required, while the sort key is optional but useful for establishing one-to-many relationships and facilitating in-range queries.

Sometimes you need to query data using the primary key, and sometimes you need to query by an attribute that is not your primary/secondary key or sort key. To tackle this issue, DynamoDB provides two kinds of indexes:

* **Local Secondary Index (LSI)** - Let’s say you want to find out all fulfilled orders. You would have to query for all orders and then look for fulfilled ones in the results, which is not very efficient with large tables. But you have LSIs to help us out. You can create an LSI with the same primary key (order\_ID) and a different secondary key (fulfilled). Now your query can be based on the key of the LSI. This is fast and efficient. The LSI is located on the same partition as the item in the table, ensuring consistency. Whenever an item is updated, the corresponding LSI is also updated and acknowledged.

The same primary key is used to partition both the LSI and the parent table, even though the parent table can have a different sort key. In the index, you can choose to have just the keys or other attributes projected or include all attributes – depending on what attributes you want to be returned with the query. There is a limit of 10 GB on LSI storage as it uses the partition storage of the original table.

* **Global Secondary Index (GSI)** -A GSI is an extension of the concept of indexes, allowing for more complex queries with various attributes as query criteria. In some cases, using the existing primary/sort key does not suffice. To address this, you can define a GSI as a parallel or secondary table with a partition key that is different from the original table and an alternate sort key. When creating a GSI, you must specify the expected workload for read and write capacity units. Similar to an LSI, you can choose to have just the keys or other attributes projected or include all attributes – depending on what attributes you want to be returned with the query.

The followings are the key differences between an LSI and GSI:

|  |  |
| --- | --- |
| **Local Secondary Index (LSI)** | **Global Secondary Index (GSI)** |
| You have to define an LSI upfront, and it can be created during table creation only. | You can define a GSI at any time, even after table creation. |
| LSI shares WCU/RCU with the main table, so you must have enough read/write capacity to accommodate LSI need. | GSI WCU/RCU is independent of the table, so it can scale without impacting the main table. |
| LSI size is a maximum of 10 GB in sync with the primary table partition, which is limited to 10 GB size per partition. | As GSI is independent of the main table, so it has no size limits. |
| You can create a maximum of 5 LSIs. | You can create up to 20 GSIs. |
| As LSIs tie up to the main table, they offer “strong consistency,” which means their current access to the most updated data. | GSI offer “eventual consistency,” which means there may be a slight lag on data updates and lowers the chances you get stable data. |

*Table 7.3: DynamoDB Index – LSI vs. GSI*

So, you may ask when to use an LSI vs. GSI. In a nutshell, a GSI is more flexible. An LSI is useful when you want to query data based on an alternate sort key within the same partition key as the base table.

It allows you to perform fast, efficient queries with minimal latency, and is best suited for scenarios where you know the specific queries that will be performed on your data. A GSI, on the other hand, allows you to query data based on attributes that are not part of the primary key or sort key of the base table. It’s useful when you want to perform ad-hoc queries on different attributes of the data or when you need to support multiple access patterns. A GSI can be used to scale read queries beyond the capacity of the base table and can also be used to query data across partitions.

In general, if your data size is small enough, and you only need to query data based on a different sort key within the same partition key, you should use an LSI. If your data size is larger, or you need to query data based on attributes that are not part of the primary key or sort key, you should use a GSI. However, keep in mind that a GSI comes with some additional cost and complexity in terms of provisioned throughput, index maintenance, and eventual consistency.

If an item collection’s data size exceeds 10 GB, the only option is to use a GSI as an LSI limits the data size in a particular partition. If eventual consistency is acceptable for your use case, a GSI can be used as it is suitable for 99% of scenarios.

DynamoDB is very useful in designing serverless event-driven architecture. You can capture the item-level data change, e.g., putItem, updateItem, and delete, by using DynamoDB Streams. You can learn more about Amazon DynamoDB by visiting the AWS page: <https://aws.amazon.com/dynamodb/>.

You may want a more sophisticated database when storing JSON documents for index and fast search. Let’s learn about the AWS document database offering.

**Amazon DocumentDB**

In this section, let’s talk about Amazon DocumentDB, *“a fully managed MongoDB-compatible database service designed from the ground up to be fast, scalable, and highly available”*. DocumentDB is a purpose-built document database engineered for the cloud that provides millions of requests per second with millisecond latency and can scale-out to 15 read replicas in minutes. It is compatible with MongoDB 3.6/4.0 and managed by AWS, which means no hardware provisioning, auto-patching, quick setup, good security, and automatic backups are needed.

If you look at the evolution of document databases, what was the need for document databases in this new world? At some point, JSON became the de facto standard for data interchange and data modeling within applications. Using JSON in the application and then trying to map JSON to relational databases introduced friction and complication.

**Object Relational Mappers** (**ORMs**) were created to help with this friction, but there were complications with performance and functionality. A crop of document databases popped up to solve the problem. The need for DocumentDB is primarily driven by the following reasons:

* Data is stored in documents that are in a JSON-like format, and these documents are considered as first-class objects within the database. Unlike traditional databases where documents are stored as values or data types, in DocumentDB, documents are the key design point of the database.
* Document databases offer flexible schemas, making it easy to represent hierarchical and semi-structured data. Additionally, powerful indexing capabilities make querying such documents much faster.
* Documents naturally map to object-oriented programming, which simplifies the flow of data between your application and the database.
* Document databases come with expressive query languages that are specifically built for handling documents. These query languages enable ad hoc queries and aggregations across documents, making it easier to extract insights from data.

Let’s take user profiles, for example; let’s say that Jane Doe plays a new game called ExplodingSnails, you can easily add that information to her profile, and you don’t have to design a complicated schema or create any new tables – you simply add a new set of fields to your document. Similarly, you can add an array of Jane’s promotions. The document model enables you to evolve applications quickly over time and build applications faster.

In the case of a document database, records contain structured or semi-structured values. This structured or semi-structured data value is called a document. It is typically stored using **Extensible Markup Language** (**XML**), **JavaScript Object Notation** (**JSON**), or **Binary JavaScript Object Notation** (**BSON**) format types.

What are document databases suitable for?

* Content management systems
* E-commerce applications
* Analytics
* Blogging applications

When are they not appropriate?

* Requirements for complex queries or table joins
* OLTP applications

**Advantages of DocumentDB**

Compared to traditional relational databases, Amazon DocumentDB offers several advantages, such as:

* **On-demand instance pricing**: You can pay by the hour without any upfront fees or long-term commitments, which eliminates the complexity of planning and purchasing database capacity ahead of time. This pricing model is ideal for short-lived workloads, development, and testing.
* **Compatibility with MongoDB 3.x and 4.x**: Amazon DocumentDB supports MongoDB 3.6 drivers and tools, allowing customers to use their existing applications, drivers, and tools with little or no modification. By implementing the Apache 2.0 open source MongoDB 4.x API on a distributed, fault-tolerant, self-healing storage system, Amazon DocumentDB offers the performance, scalability, and availability necessary for operating mission-critical MongoDB workloads at scale.
* **Migration support**: You can use the AWS **Database Migration Service** (**DMS**) to migrate MongoDB databases from on-premises, or on Amazon EC2, to Amazon DocumentDB at no additional cost (for up to six months per instance) with minimal downtime. DMS allows you to migrate from a MongoDB replica set or a sharded cluster to Amazon DocumentDB.
* **Flexible schema**: DocumentDB has a flexible schema, which means that the structure of the documents within the database can vary. This can be useful in situations where the data being stored has a complex or hierarchical structure, or when the data being stored is subject to frequent changes.
* **High performance**: DocumentDB is designed for high performance and can be well suited for applications that require fast read and write access to data.
* **Scalability**: DocumentDB is designed to be horizontally scalable, which means that it can be easily expanded to support large amounts of data and a high number of concurrent users.
* **Easy querying**: Document DB provides powerful and flexible query languages that make it easy to retrieve and manipulate data within the database.

DocumentDB has recently introduced new features that allow for ACID transactions across multiple documents, statements, collections, or databases. You can learn more about Amazon DocumentDB by visiting the AWS page: <https://aws.amazon.com/documentdb/>.

If you want sub-millisecond performance, you need your data in memory. Let’s learn about in-memory databases.

**In-memory database**

Studies have shown that if your site is slow, even for just a few seconds, you will lose customers. A slow site results in 90% of your customers leaving the site. 57% of those customers will purchase from a similar retailer, and 25% of them will never return. These statistics are from a report by Business News Daily (<https://www.businessnewsdaily.com/15160-slow-retail-websites-lose-customers.html>). Additionally, you will lose 53% of your mobile users if a page load takes longer than 3 seconds (<https://www.business.com/articles/website-page-speed-affects-behavior/>).

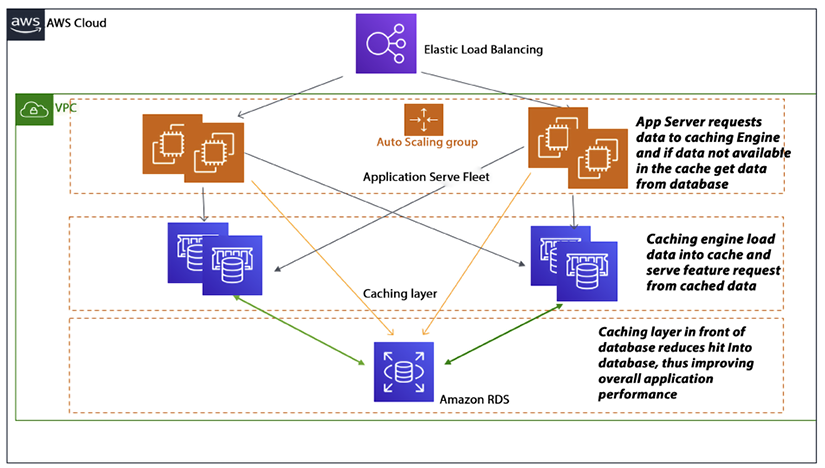
We’re no longer in the world of thinking our users are okay with a few seconds of wait time. These are demanding times for services, and to keep up with demand, you need to ensure that users aren’t waiting to purchase your service, products, and offerings to continue growing your business and user base. There is a direct link between customers being forced to wait, and a loss of revenue.

Applications and databases have changed dramatically not only in the past 50 years but also just in the past 10. Where you might have had a few thousand users who could wait for a few seconds for an application to refresh, now you have microservices and IoT devices sending millions of requests per second across the globe that require immediate responses. These changes have caused the application and database world to rethink how data is stored and accessed. It’s essential to use the right tool for the job. In-memory data stores are used when there is a need for maximum performance. This is achieved through extremely low latency per request and incredibly high throughput as you are caching data in memory, which helps to increase the performance by taking less time to read from the original database. Think of this as high-velocity data.

**In-memory databases**, or **IMDBs** for short, usually store the entire data in the main memory. Contrast this with databases that use a machine’s RAM for optimization but do not store all the data simultaneously in primary memory and instead rely on disk storage. IMDBs generally perform better than disk-optimized databases because disk access is slower than direct memory access. In-memory operations are more straightforward and can be performed using fewer CPU cycles. In-memory data access seeks time when querying the data, which enables faster and more consistent performance than using long-term storage. To get an idea of the difference in performance, in-memory operations are usually measured in nanoseconds, whereas operations that require disk access are usually measured in milliseconds. So, in-memory operations are usually about a million times faster than operations needing disk access.

Some use cases of in-memory databases are real-time analytics, chat apps, gaming leaderboards, and caching. AWS provides Amazon ElastiCache to fulfill in-memory data caching needs.

As shown in the following diagram, based on your *data access pattern*, you can use either **lazy caching** or **write-through**. In lazy caching, the cache engine checks whether the data is in the cache and, if not, gets it from the database and keeps it in the cache to serve future requests. Lazy caching is also called the **cache aside pattern**.



*Figure 7.4: Application caching pattern architecture*

You can see the caching flow in the above diagram, where the first app server sends a data request to the caching engine, which tries to load data from the cache. If data is not available in the cache, then the cache engine goes to the database and loads the required data into the cache. The Amazon ElastiCache service is an AWS-provided cache database. Let’s learn more details about it.

**Amazon ElastiCache**

Amazon ElastiCache is a cloud-based web service that enables users to deploy and manage an in-memory cache with ease. By storing frequently accessed data in memory, in-memory caches can enhance application performance, enabling faster data access compared to retrieving data from a slower backing store such as a disk-based database. ElastiCache offers support for two popular open-source in-memory cache engines: Memcached and Redis. Both engines are well known for their reliability, scalability, and performance.

With ElastiCache, you can quickly and easily set up, manage, and scale an in-memory cache in the cloud.

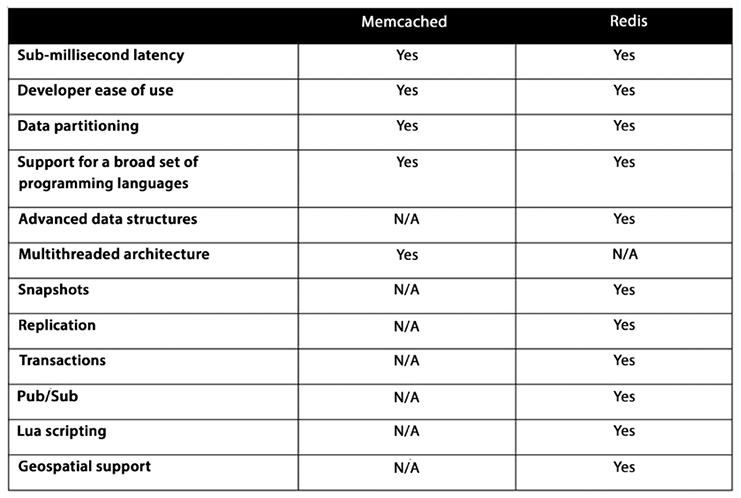
ElastiCache automatically handles tasks such as provisioning cache nodes, configuring cache clusters, and monitoring the health of the cache environment, allowing you to focus on developing and deploying your application.

In addition, ElastiCache integrates seamlessly with other AWS services, making it easy to use the cache as a caching layer for other services like Amazon RDS or Amazon DynamoDB. This can help improve the performance and scalability of your overall application architecture.

Amazon ElastiCache enables users of the service to configure, run, and scale an IMDB and to build data-intensive applications. In-memory data storage boosts applications’ performance by retrieving data directly from memory.

**Redis versus Memcached**

Since ElastiCache offers two flavors of in-memory databases, the obvious question is, which one is better? From our research, the answer now appears to be Redis, unless you are already a heavy user of Memcached. If your organization already has committed to Memcached, it is likely not worth porting it to Redis. But for new projects, the better option is Redis. This could change in the future, but as of this writing, Redis continues to gain supporters. Here is a comparison of the features and capabilities of the two:



*Figure 7.5: Comparison of Redis and Memcached*

The preceding table shows the key differences between Redis and Memcached. You can choose by validating your options, and you can learn more about Amazon ElastiCache by visiting the AWS page: <https://aws.amazon.com/elasticache/>.

AWS recently launched Amazon MemoryDB for Redis due to Redis’s popularity. It is a durable, in-memory database service that provides ultra-fast performance and is compatible with Redis version 6.2. This service is designed explicitly for modern, microservice-based applications, and it offers Redis’s flexible data structures, APIs, and commands. With Amazon MemoryDB, all data is stored in memory, allowing for microsecond read and single-digit millisecond write latency, as well as high throughput. You can learn more about Amazon MemoryDB by visiting the AWS page: <https://aws.amazon.com/memorydb/>.

Now data is getting more complicated with many-to-many relationships and several layers in social media. You need a specific database to drive the relationship, such as friends of friends and their common likes. Let’s look at graph databases to solve this problem.

**Graph databases**

Graph databases are data stores that treat relationships between records as first-class citizens. In traditional databases, relationships are often an afterthought. In the case of relational databases, relationships are implicit and manifest themselves as foreign key relationships. In graph databases, relationships are explicit, significant, and optimized using graph database language; these relationships are called edges.

In some aspects, graph databases are similar to NoSQL databases. They are also schema-less. For certain use cases, they offer much better data retrieval performance than traditional databases. As you can imagine, graph databases are particularly suited for use cases that place heavy importance on relationships among entities.

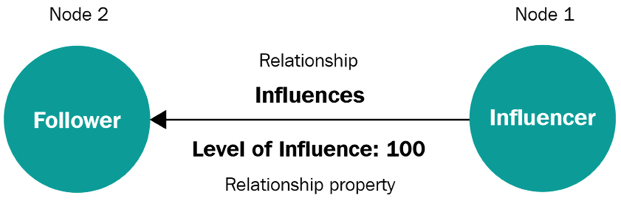
Accessing data nodes and edges in a graph database is highly efficient. It usually can occur in constant time. With graph databases, it is not uncommon to be able to traverse millions of edges per second.

Graph databases can handle nodes with many edges regardless of the dataset’s number of nodes. You only need a pattern and an initial node to traverse a graph database. Graph databases can easily navigate the adjacent edges and nodes around an initial starting node while caching and aggregating data from the visited nodes and edges. As an example of a pattern and a starting point, you might have a database that contains ancestry information. In this case, the starting point might be you, and the pattern might be a parent.

So, in this case, the query would return the names of both of your parents. The following are the components of a graph database:

* **Nodes**: Nodes are elements or entities in a graph. They contain a series of properties, attributes, or key-value pairs. Nodes can be given tags, which constitute roles in the domain. Node labels can be employed to assign metadata (such as indices or constraints) to the nodes.
* **Edges**: Edges supply directed, named, and semantically significant connections between two nodes. An edge has a direction, a type, a start node, and an end node. Like a node, an edge can also have properties. In some situations, an edge can have quantitative properties, such as weight, cost, and strength. Due to the efficient way an edge is stored, two nodes can share edges regardless of the quantity or type without a performance penalty. Edges have a direction, but edges can be traversed efficiently in both directions.

The following diagram shows the relationship between “Follower” and “Influencer” in a social media application, where the nodes depict the entity type (Follower and Influencer) and the edge (Influences) shows their relationship with their level of influence as the property weight. Here, the level of influence is 100, which means the Follower just started following the Influencer and with time, as they give more likes and views to the influencer’s post, their level can increase to 200, 300, or 400.



*Figure 7.6: Example of a relationship*

Two primary graph models are widely used. A property graph is a common name for an attributed, multi-relational graph. The leading property graph API is the open standard Apache TinkerPop™ project. It provides an imperative traversal language, called Gremlin, that can be used to write traversals on property graphs, and many open-source and vendor implementations support it. You may opt for property graphs to represent relational models, and the Apache TinkerPop Gremlin traversal language could be a favorable option as it offers a method to navigate through property graphs. You might also like openCypher, an open-source declarative query language for graphs, as it provides a familiar SQL-like structure to compose queries for graph data.

The second is the **Resource Description Framework** (**RDF**), standardized by the W3C in a set of standards collectively known as the Semantic Web. The SPARQL query language for RDF allows users to express declarative graph queries against RDF graph models. The RDF model is also a labeled, directed multi-graph, but it uses the concept of triples, subject, predicate, and object, to encode the graph. Now let’s look at Amazon Neptune, which is Amazon’s graph database service.

**Amazon Neptune**

Amazon Neptune is a managed service for graph databases, which uses nodes, edges, and properties to represent and store data, making it a unique type of database. This data model is well suited to represent the complex relationships found in many types of data, such as the relationships between people in a social network or the interactions between different products on an e-commerce website.

Neptune supports the property graph and W3C’s RDF standards, making it easy to integrate with other systems and tools that support these standards. Neptune also provides a query language called Gremlin which is powerful and easy to use, which makes it easy to perform complex graph traversals and data manipulation operations on the data stored in the database.

In addition, Neptune is highly scalable and available, with the ability to support billions of vertices and edges in a single graph. It is also fully managed, which means that Amazon takes care of the underlying infrastructure and performs tasks such as provisioning, patching, and backup and recovery, allowing you to focus on building and using your application. You can learn more about Amazon Neptune by visiting the AWS page: <https://aws.amazon.com/neptune/>.

A lot of data comes with a timestamp, and you need a specific database to store time-series data. Let’s learn more about it.

**Time-series databases**

A **time-series database** (**TSDB**) is a database specifically designed and optimized to store events. What is an event, you ask? It is an action that happens at a specific point in time. With events, it’s not only important to track *what* happened but just as important to track *when* it happened. The unit of measure to use for the time depends on the use case. For some applications, it might be enough to know on what day the event happened. But for other applications, it might be required to keep track of the time down to the millisecond. Some examples of projects that might benefit from a TSDB are as follows:

* Performance monitoring
* Networking and infrastructure applications
* Adtech and click stream processing
* Sensor data from IoT applications
* Event-driven applications
* Financial applications
* Log analysis
* Industrial telemetry data for equipment maintenance
* Other analytics projects

A TSDB is optimized to measure changes over time. Time series values can differ from other data types and require different optimization techniques.

Common operations in a TSDB are as follows:

* Millions of inserts from disparate sources potentially per second
* Summarization of data for downstream analytics
* Access to individual events

TSDBs are ideally suited for storing and processing IoT data. Time-series data has the following properties (which might not be present with other data types):

* The order in which the events occur may be necessary.
* Data is only inserted; it is not updated.
* Queries have a time interval component in their filters.

RDBMSes can store this data, but they are not optimized to process, store, and analyze this type of data. Amazon Timestream was purpose-built exclusively for this data type and, therefore, is much more efficient.

Do you feel comfortable about when you should use TSDBs? If you need to store events or track logs or trades, or the time and date when something happened to take center stage, then a TSDB is probably an excellent solution to your problem.

**Amazon Timestream**

Amazon Timestream is a scalable and fully managed TSDB. Amazon Timestream can persist and analyze billions of transactions per minute at about 1/10 of the cost of RDBMS equivalents. IoT devices and smart industrial machines are becoming more popular by the day. These applications generate events that need to be tracked and measured, sometimes with real-time requirements.

Amazon Timestream has an adaptive query processing engine that can make heads or tails of time-series data as it comes in by inferring data location and data format. Amazon Timestream has features that can automate query rollups, retention, tiering, and data compression. Like many other Amazon services, Timestream is serverless, so it can automatically scale up or down depending on how much data is coming into the streams. Also, because it’s serverless and fully managed, tasks such as provisioning, operating system patching, configuration, backups, and tiering are not the responsibility of the DevOps team, allowing them to focus on more important tasks.

Timestream enables you to store multiple measures in a single table row with its multi-measure records feature, instead of one measure per table row, making it easier to migrate existing data from relational databases to Timestream with minimal changes. Scheduled computations are also available, allowing you to define a computation or query and its schedule. Timestream will automatically and periodically run the queries and store the results in a separate table. Additionally, Timestream automatically determines whether data should be written to the memory or magnetic store based on the data’s timestamp and configured data retention window, thereby reducing costs. You can learn more about Amazon Timestream by visiting the AWS page: <https://aws.amazon.com/timestream/>.

You often want to make your database tamper-proof and keep a close record of any activity. To serve this purpose, let’s learn about ledger databases.

**Ledger databases**

A **ledger database** (**LDB**) is a database that delivers a cryptographically verifiable, immutable, and transparent transaction log orchestrated by a central authority:

* **LDB immutability**: Imagine you deposit $1,000 in your bank. You see on your phone that the deposit was carried out, and it now shows a balance of $1,000. Then imagine you recheck it tomorrow, and this time, it says $500. You would not be too pleased, would you? The bank needs to ensure that the transaction is immutable and that no one can change it after the fact. In other words, only inserts are allowed, and updates cannot be performed. This assures that transactions cannot be changed once they are persisted.
* **LDB transparency**: In this context, transparency refers to the ability to track changes to the data over time. The LDB should be able to keep an audit log. This audit log, at a minimum, should include who changed the data, when the data was changed, and what the value of the data was before it was changed.
* **LDB cryptographic verifiability**: How can we ensure that our transaction will be immutable? Even though the database might not support update operations, what’s stopping someone from using a backdoor and updating the record? If we use cryptography when the transaction is recorded, the entire transaction data is hashed. In simple terms, the string of data that forms the transaction is whittled down into a smaller string of unique characters. Whenever the transaction is hashed, it needs to match that string. In the ledger, the hash comprises the transaction data and appends the previous transaction’s hash. Doing this ensures that the entire chain of transactions is valid. If someone tried to enter another transaction in between, it would invalidate the hash, and it would detect that the foreign transaction was added via an unauthorized method.

The prototypical use case for LDBs is bank account transactions. If you use an LDB for this case, the ledger records all credits and debits related to the bank account. It can then be followed from a point in history, allowing us to calculate the current account balance. With immutability and cryptographic verification, we are assured that the ledger cannot be deleted or modified. With other methods, such as RDBMSes, all transactions could be changed or erased.

**Amazon Quantum Ledger Database (QLDB)**

Amazon QLDB is a fully managed service that provides a centralized trusted authority to manage an immutable, transparent, and cryptographically verifiable ledger. QLDB keeps track of application value changes and manages a comprehensive and verifiable log of changes to the database.

Historically, ledgers have been used to maintain a record of financial activities. Ledgers can keep track of the history of transactions that need high availability and reliability. Some examples that need this level of reliability are as follows:

* Financial credits and debits
* Verifying the data lineage of a financial transaction
* Tracking the location history of inventory in a supply chain network

Amazon QLDB offers various blockchain services, such as anonymous data sharing and smart contracts, while still using a centrally trusted transaction log.

QLDB is designed to act as your system of record or source of truth. When you write to QLDB, your transaction is committed to the journal. The journal is what provides immutability through append-only interactions and verifiability through cryptographic hashing. QLDB treats all interactions, such as reads, inserts, updates, and deletes, like a transaction and catalogs everything sequentially in this journal.

Once the transaction is committed to the journal, it is immediately materialized into tables and indexes. QLDB provides a current state table and indexed history as a default when you create a new ledger. Leveraging these allows customers to, as the names suggest, view the current states of documents and seek out specific documents and their revisions easily. You can learn more about Amazon QLDB by visiting the AWS page: <https://aws.amazon.com/qldb/>.

Sometimes you need a database for large-scale applications that need fast read and write performance, which a wide-column store can achieve. Let’s learn more about it.

**Wide-column store databases**

Wide-column databases can sometimes be referred to as column family databases. A wide-column database is a NoSQL database that can store petabyte-scale amounts of data. Its architecture relies on persistent, sparse matrix, multi-dimensional mapping using a tabular format. Wide-column databases are generally not relational.

When is it a good idea to use wide-column databases?

* Sensor logs and IoT information
* Geolocation data
* User preferences
* Reporting
* Time-series data
* Logging applications
* Many inserts, but not many updates
* Low latency requirements

When are wide-column databases not a good fit? They are good when the use case calls for ad hoc queries:

* Heavy requirement for joins
* High-level aggregations
* Requirements change frequently
* OLTP uses cases

Apache Cassandra is probably the most popular wide-column store implementation today. Its architecture allows deployments without single points of failure. It can be deployed across clusters and data centers.

**Amazon Keyspaces** (formerly Amazon Managed Apache Cassandra Service, or Amazon MCS) is a fully managed service that allows users to deploy Cassandra workloads. Let’s learn more about it.

**Amazon Keyspaces (for Apache Cassandra)**

Amazon Keyspaces (formerly known as Amazon Cassandra) is a fully managed, scalable, and highly available NoSQL database service. NoSQL databases are a type of database that does not use the traditional table-based relational database model and is well suited for applications that require fast, scalable access to large amounts of data.

Keyspaces is based on Apache Cassandra, an open-source NoSQL database that is widely used for applications that require high performance, scalability, and availability. Keyspaces provides the same functionality as Cassandra, with the added benefits of being fully managed and integrated with other AWS services.

Keyspaces supports both table and **Cassandra Query Language** (**CQL**) APIs, making it easy to migrate existing Cassandra applications to Keyspaces. It also provides built-in security features, such as encryption at rest and network isolation, using Amazon VPC and integrates seamlessly with other AWS services, such as Amazon EMR and Amazon SageMaker.

Servers are automatically spun up or brought down, and, as such, users are only charged for the servers Cassandra is using at any one time. Since AWS manages it, users of the service never have to provision, patch, or manage servers, and they don’t have to install, configure, or tune software. Cassandra in AWS can be configured to support thousands of user requests per second.

Cassandra is a NoSQL database; as expected, it doesn’t support SQL. The query language for Cassandra is the CQL. The quickest method to interact with Apache Cassandra is using the CQL shell, which is called **cqlsh**. With cqlsh, you can create tables, insert data into the tables, and access the data via queries, among other operations.

Keyspaces supports the Cassandra CQL API. Because of this, the current code and drivers developed in Cassandra will work without changing the code. Using Amazon Keyspaces instead of just Apache Cassandra is as easy as modifying your database endpoint to point to an Amazon MCS service table.

In addition to Keyspaces being wide-column, the major difference from DynamoDB is that it supports composite partition keys and multiple clustering keys, which are not available in DynamoDB. However, DynamoDB has better connectivity with other AWS services, such as Athena, Kinesis, and Elasticsearch.

Keyspaces provides an SLA for 99.99% availability within an AWS Region. Encryption is enabled by default for tables, and tables are replicated three times in multiple AWS Availability Zones to ensure high availability. You can create continuous backups of tables with hundreds of terabytes of data with no effect on your application’s performance, and recover data to any point in time within the last 35 days. You can learn more about Amazon Keyspaces by visiting the AWS page: <https://aws.amazon.com/keyspaces/>.

**Benefits of AWS database services**

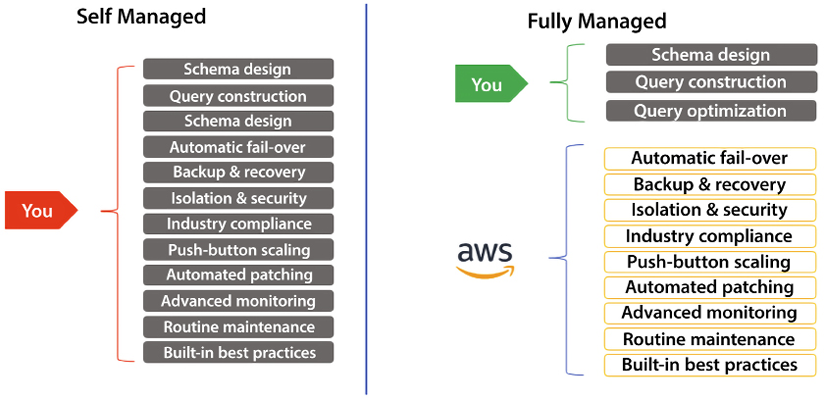
In the new world of cloud-born applications, a one-size-fits-all database model no longer works. Modern organizations will not only use multiple types of databases for multiple applications, but many will use multiple types of databases in a single application. To get more value from data, you can choose the following three options available in AWS based on your workload.

**Moving to fully managed database services**

Managing and scaling databases in a legacy infrastructure, whether on-premises or self-managed in the cloud (on EC2), can be a tedious, time-consuming, and costly process. You have to worry about operational efficiency issues such as:

* The process of installing hardware and software, configuring settings, patching, and creating backups can be time-consuming and laborious
* Performance and availability issues
* Scalability issues, such as capacity planning and scaling clusters for computing and storage
* Security and compliance issues, such as network isolation, encryption, and compliance programs, including PCI, HIPAA, FedRAMP, ISO, and SOC

Instead of dealing with the challenges mentioned above, you would rather spend your time innovating and creating new applications instead of managing infrastructure. With AWS-managed databases, you can avoid the need to over- or under-provision infrastructure to accommodate application growth, spikes, and performance requirements, as well as the associated fixed capital costs such as software licensing, hardware refresh, and maintenance resources. AWS manages everything for you, so you can focus on innovation and application development, rather than infrastructure management. You won’t need to worry about administrative tasks such as server provisioning, patching, setup, configuration, backups, or recovery. AWS continuously monitors your clusters to ensure your workloads are running with self-healing storage and automated scaling, allowing you to focus on higher-value tasks such as schema design and query optimization.



*Figure 7.7: Fully managed database services on AWS*

Let’s look at the next benefit of using a purpose-built database. With purpose-built databases, your application doesn’t need a one-fit-for-all architecture and it doesn’t have to be molded to accommodate a decade-old relational database. Purpose-built databases help you to achieve maximum output and performance as per the nature of your application.

**Building modern applications with purpose-built databases**

In the 60s and 70s, mainframes were the primary means of building applications, but by the 80s, client-server architecture was introduced and significantly changed application development. Applications became more distributed, but the underlying data model remained mostly structured, and the database often functioned as a monolith. With the advent of the internet in the 90s, three-tier application architecture emerged. Although client and application code became more distributed, the underlying data model continued to be mainly structured, and the database remained a monolith. For nearly three decades, developers typically built applications against a single database. And that is an interesting data point because if you have been in the industry for a while, you often bump into folks whose mental model is, “Hey, I’ve been building apps for a long time, and it’s always against this one database.”

But what has changed? Fast forward to today, and microservice architectures are how applications are built in the cloud. Microservices have now extended to databases, providing developers with the ability to break down larger applications into smaller, specialized services that cater to specific tasks.

This allows developers to choose the best tool for the job instead of relying on a single, overworked database with a single storage and compute engine that struggles to handle every access pattern.

Today’s developers and end-users have different expectations compared to the past. They require lower latency and the ability to handle millions of transactions per second with many concurrent users. As a result, data management systems have evolved to include specialized storage and compute layers optimized for specific use cases and workloads. This allows developers to avoid trade-offs between functionality, performance, and scale.

Plus, what we’ve seen over the last few years is that more and more companies are hiring technical talent in-house to take advantage of the enormous wave of technological innovation that the cloud provides. These developers are building not in the monolithic ways of the past but with microservices, where they compose the different elements together using the right tool for the right job. This has led to the highly distributed, loosely coupled application architectures we see powering the most demanding workloads in the cloud.

Many factors contribute to the performance, scale, and availability requirements of modern apps:

* **Users** – User growth is a common KPI for businesses, and the cloud’s global reach enables businesses to touch millions of new customers.
* **Data volume** – Businesses are capturing more data about their customers to either increase the value of their existing products or sell them new products. This results in terabyte- or even petabyte-scale data.
* **Locality** – Businesses are often expanding their presence into new markets to reach new users, which complicates the architectures of their solutions/products.
* **Performance** – Businesses don’t want to dilute the user experience to reach new markets or grow their customer base. If customers find a better alternative, they’ll use it.
* **Request rate** – As businesses use the cloud’s global reach to develop more interactive user experiences in more markets, they need their apps and databases to handle unprecedented levels of throughput.
* **Access/scale** – As businesses embark on their digital transformations, these scale measures are compounded by the number of devices that access their applications. There are billions of smartphones worldwide, and businesses connect smartphones, cars, manufacturing plants, devices in our homes, and more to the cloud. This means many, many billions of devices are connected to the cloud.
* **Economics** – Businesses can’t invest millions of dollars in hardware and licenses up front and hope they’ll succeed, that model is unrealistic in 2023. Instead, they have to hedge their success by only paying for what they use, without capping how much they can grow. These dynamics combined have changed how businesses build applications. These modern applications have wildly different database requirements, which are more advanced and nuanced than simply running everything in a relational database.

The traditional approach of using a relational database as the sole data store for an application is no longer sufficient. To address this, developers are leveraging their expertise in breaking down complex applications into smaller components and choosing the most appropriate tool for each task. This results in well-architected applications that can scale effectively. The optimal tool for a given task often varies by use case, leading developers to build highly distributed applications using multiple specialized databases.

Now you have learned about the different types of AWS databases, let’s go into more detail about moving on from legacy databases.

**Moving on from legacy databases**

Numerous legacy applications have been developed on conventional databases, and consumers have had to grapple with database providers that are expensive, proprietary, and impose punishing licensing terms and frequent audits. Oracle, for instance, announced that they would double licensing fees if their software is run on AWS or Microsoft. As a result, customers are attempting to switch as soon as possible to open-source databases such as MySQL, PostgreSQL, and MariaDB.

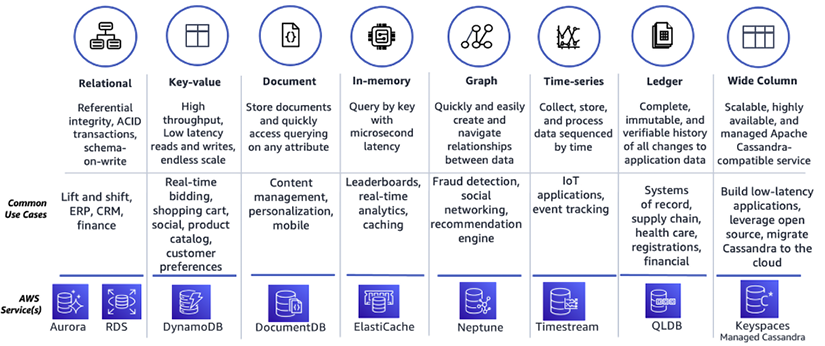
Customers who are migrating to open-source databases are seeking to strike a balance between the pricing, freedom, and flexibility of open-source databases and the performance of commercial-grade databases. Achieving the same level of performance on open-source databases as on commercial-grade databases can be challenging and necessitates a lot of fine-tuning.

AWS introduced Amazon Aurora, a cloud-native relational database that is compatible with MySQL and PostgreSQL to address this need. Aurora aims to provide a balance between the performance and availability of high-end commercial databases and the simplicity and cost-effectiveness of open-source databases. It boasts 5 times better performance than standard MySQL and 3 times better performance than standard PostgreSQL, while maintaining the security, availability, and reliability of commercial-grade databases, all at a fraction of the cost. Additionally, customers can migrate their workloads to other AWS services, such as DynamoDB, to achieve application scalability.

In this section, you learned about the benefits of AWS database services. Now, there are so many database services that you have learned about, so let’s put them together and learn how to choose the right database.

**Choosing the right tool for the job**

In the previous sections, you learned how to classify databases and the different database services that AWS provides. In a nutshell, you learned about the following database services under different categories:



*Figure 7.8: AWS database services in a nutshell*

When you think about the collection of databases shown in the preceding diagram, you may think, “Oh, no. You don’t need that many databases. I have a relational database, and it can take care of all this for you”. Swiss Army knives are hardly the best solution for anything other than the most straightforward task. If you want the right tool for the right job that gives you the expected performance, productivity, and customer experience, you want a unified purpose-built database. So no one tool rules the world, and you should have the right tool for the right job to make you spend less money, be more productive, and change the customer experience.

Consider focusing on common database categories to choose the right database instead of browsing through hundreds of different databases. One such category is ‘relational,’ which many people are familiar with. Suppose you have a workload where strong consistency is crucial, where you will collaborate with the team to define schemas, and you need to figure out every single query that will be asked of the data and require consistent answers. In that case, a relational database is a good fit.

Popular options for this category include Amazon Aurora, Amazon RDS, open-source engines like PostgreSQL, MySQL, and MariaDB, as well as RDS commercial engines such as SQL Server and Oracle Database.

AWS has developed several purpose-built non-relational databases to facilitate the evolution of application development. For instance, in the key-value category, Amazon DynamoDB is a database that provides optimal performance for running key-value pairs at a single-digit millisecond latency and at a large scale. On the other hand, if you require a flexible method for storing and querying data in the same document model used in your application code, then Amazon DocumentDB is a suitable choice. This document database is designed to handle JSON documents in their natural format efficiently and can be scaled effortlessly.

Do you recall the era of XML? XML 1.0 was established in 1998. Commercial systems then added an XML data type to become an XML database. However, this approach had limitations, as many database operators needed help working with that data type. Today, document databases have replaced XML databases. Amazon DocumentDB, launched in January 2019, is an excellent example of such a database.

If your application requires faster response times than single-digit millisecond latency, consider an in-memory database and cache that can access data in microseconds. Amazon ElastiCache offers management for Redis and Memcached, making it possible to retrieve data rapidly for real-time processing use cases such as messaging, and real-time geospatial data such as drive distance.

Suppose you have large datasets with many connections between them. For instance, a sports company should link its athletes with its followers and provide personalized recommendations based on the interests of millions of users. Managing all these connections and providing fast queries can be challenging with traditional relational databases. In this case, you can use Amazon Neptune, a graph database designed to efficiently handle complex queries with interconnected data.

Time-series data is not just a timestamp or a data type that you might use in a relational database. Instead, a time-series database’s core feature is that the primary axis of the data model is time. This allows for the optimization of data storage, scaling, and retrieval. Amazon Timestream is an example of a purpose-built time-series database that provides fast and scalable querying of time-series data.

Amazon QLDB is a fully managed ledger database service. A ledger is a type of database that is used to store and track transactions and is typically characterized by its immutability and the ability to append data in sequential order.

QLDB is a transactional database that uses an immutable, append-only ledger to store data. This means that once data is written to the ledger, it cannot be changed, and new data can only be added in sequential order. This makes QLDB well suited for applications that require a transparent, auditable, and verifiable record of transactions.

A wide-column database is an excellent choice for applications that require fast data processing with low latency, such as industrial equipment maintenance, trade monitoring, fleet management, and route optimization. Amazon Keyspaces for Apache Cassandra provides a wide-column database option that allows you to develop applications that can handle thousands of requests per second with practically unlimited throughput and storage.

You should spend a significant amount of time clearly articulating the business problem you are trying to solve. Some of the questions the requirements should answer are as follows:

* How many users are expected?
* How many transactions per day will occur?
* How many records need to be stored?
* Will there be more writes or reads?
* How will the data need to be accessed (only by primary key, by filtering, or some other way)?

Why are these questions important? SQL has served us well for several decades now, as it is pervasive, and has a lot of mindshare. So, why would we use anything else? The answer is performance. In instances where there is a lot of data and it needs to be accessed quickly, NoSQL databases might be a better solution. SQL vendors realize this and are constantly trying to improve their offerings to better compete with NoSQL, including adopting techniques from the NoSQL world. For example, Aurora is a SQL service, and it now offers Aurora Serverless, taking a page out of the NoSQL playbook.

As services get better, the line between NoSQL and SQL databases keeps on blurring, making the decision about what service to use more and more difficult. Depending on your project, you might want to draw up a Proof of Concept using a couple of options to determine which option performs better and fits your needs better.

Another reason to choose SQL or NoSQL might be the feature offered by NoSQL to create schema-less databases. Creating databases without a schema allows for fast prototyping and flexibility. However, tread carefully. Not having a schema might come at a high price.

Allowing users to enter records without the benefit of a schema may lead to inconsistent data, which becomes too variable and creates more problems than it solves. Just because we can create databases without a schema in a NoSQL environment, we should not forgo validation checks before creating a record. If possible, a validation scheme should be implemented, even when using a NoSQL option.

It is true that going schema-less increases implementation agility during the data ingestion phase. However, it increases complexity during the data access phase. So, make your choice by making a required trade-off between data context vs. data performance.

**Migrating databases to AWS**

If you find it challenging to maintain your relational databases as they scale, consider switching to a managed database service such as Amazon RDS or Amazon Aurora. With these services, you can migrate your workloads and applications without the need to redesign your application, and you can continue to utilize your current database skills.

Consider moving to a managed relational database if:

* Your database is currently hosted on-premises or in EC2.
* You want to reduce the burden of database administration and allocate DBA resources to application-centric work.
* You prefer not to rearchitect your application and wish to use the same skill sets in the cloud.
* You need a straightforward path to a managed service in the cloud for database workloads.
* You require improved performance, availability, scalability, and security.

Self-managed databases like Oracle, SQL Server, MySQL, PostgreSQL, and MariaDB can be migrated to Amazon RDS using the lift and shift approach. For better performance and availability, MySQL and PostgreSQL databases can be moved to Amazon Aurora, which offers 3-5 times better throughput. Non-relational databases like MongoDB and Redis are popularly used for document and in-memory databases in use cases like content management, personalization, mobile apps, catalogs, and real-time use cases such as caching, gaming leaderboards, and session stores. To maintain non-relational databases at scale, organizations can move to a managed database service like Amazon DocumentDB for self-managed MongoDB databases or Amazon ElastiCache for self-managed in-memory databases like Redis. These services provide a straightforward solution to manage the databases without rearchitecting the application and enable the same DB skill sets to be leveraged while migrating workloads and applications.

As you understand the different choices of databases, then the question comes of how to migrate your database to the cloud; there are five types of database migration paths available in AWS:

* **Self-service** - For many migrations, the self-service path using the DMS and **Schema Conversion Tool** (**SCT**) offers the tools necessary to execute with over 250,000 migrations completed through DMS, customers have successfully migrated their instances to AWS. Using the **Database Migration Service** (**DMS**), you can make homogeneous migrations from your legacy database service to a managed service on AWS, such as from Oracle to RDS Oracle. Alternatively, by leveraging DMS and the SCT, heterogeneous conversions are possible, such as converting from SQL Server to Amazon Aurora. The **Schema Conversion Tool** (**SCT**) assesses the source compatibility and recommends the best target engine.
* **Commercially licensed to aws databases** - This type of migration is best for customers looking to move away from the licensing costs of commercial database vendors and avoid vendor lock-in. Most of these migrations have been from Oracle and SQL Server to open-source databases and Aurora, but there are use cases for migrating to NoSQL databases as well. For example, an online store may have started on a commercial or open-source database but now is growing so fast that it would need a NoSQL database like DynamoDB to scale to millions of transactions per minute. Refactoring, however, typically requires application changes and takes more time to migrate than the other migration methods. AWS provides a Database Freedom program to assist with such migration. You can learn more about the AWS Database Freedom program by visiting the AWS page: <https://aws.amazon.com/solutions/databasemigrations/database-freedom/>.
* **MySQL Database Migrations** - Standard MySQL import and export tools can be used for MySQL database migrations to Amazon Aurora. Additionally, you can create a new Amazon Aurora database from an Amazon RDS for MySQL database snapshot with ease. Migration operations based on DB snapshots typically take less than an hour, although the duration may vary depending on the amount and format of data being migrated.
* **PostgreSQL Database Migrations** - For PostgreSQL database migrations, standard PostgreSQL import and export tools such as pg\_dump and pg\_restore can be used with Amazon Aurora. Amazon Aurora also supports snapshot imports from Amazon RDS for PostgreSQL, and replication with AWS DMS.
* **The AWS Data Lab** is a service that helps customers choose their platform and understand the differences between self-managed and managed services. It involves a 4-day intensive engagement between the customer and AWS database service teams, supported by AWS solutions architecture resources, to create an actionable deliverable that accelerates the customer’s use and success with database services. Customers work directly with Data Lab architects and each service’s product managers and engineers. At the end of a Lab, the customer will have a working prototype of a solution that they can put into production at an accelerated rate.

A Data Lab is a mutual commitment between a customer and AWS. Each party dedicates key personnel for an intensive joint engagement, where potential solutions will be evaluated, architected, documented, tested, and validated. The joint team will work for four days to create usable deliverables to enable the customer to accelerate the deployment of large AWS projects. After the Lab, the teams remain in communication until the projects are successfully implemented.

In addition to the above, AWS has an extensive Partner Network of consulting and software vendor partners who can provide expertise and tools to migrate your data to AWS.

**Summary**

In this chapter, you learned about many of the database options available in AWS. You started by revisiting a brief history of databases and innovation trends led by data. After that, you explored the database consistency model learning ACID vs. BASE and OLTP vs. OLAP.

You further explored different types of databases and when it’s appropriate to use each one, and you learned about the benefits of AWS databases and the migration approach. There are multiple database choices available in AWS, and you learned about making a choice to use the right database service for your workload.

In the next chapter, you will learn about AWS’s services for cloud security and monitoring.