5
Architecting Storage and Data Infrastructure

Virtually every application relies on **data**, and in data is where the real business value lies for most organizations. Knowing how to design a storage solution that best fits your data requirements for **durability**, **availability**, **consistency**, **latency**, **security**, and **compliance** is an essential infrastructure architecture skill.

In this chapter, you will learn how to choose a storage solution based on the type of data and application requirements. You will learn about different storage services within Google Cloud for relational and structured data and non-relational and unstructured data, and how to design those services for high availability and scalability. You will then learn how to approach backups, replication, and data consistency with the different services. Finally, you will get some hands-on assignments to familiarize yourself with two of the most popular GCP database offerings: **Cloud Spanner** and **Cloud Bigtable**.

In this chapter, we're going to cover the following main topics:

- Choosing the right storage solution
- Using relational and structured data stores
- Using non-relational and unstructured datastores

Technical requirements

For the hands-on activities in this chapter, you will need a billing-enabled Google Cloud account and optionally the Google Cloud SDK installed (if you don't have it, you can use Cloud Shell from a browser). Helper scripts and code for this chapter can be found at `https://github.com/PacktPublishing/Architecting-Google-Cloud-Solutions/tree/master/ch5`.

Choosing the right storage solution

A fundamental skill for any cloud architect is to know how to choose the right storage solution for the various types of data an organization possesses. In this section, you will start by learning a mental framework that will make it easier for you to make the right choice of data solution. Let's start by understanding and identifying the different types of data that exist.

Types of data

Data can be categorized in a few different "dimensions," so let's look at each one of them separately.

Relational versus non-relational

This first distinction applies to whether or not datasets are organized according to the **relational model** for databases. A collection of tables of data is considered relational when the relationship between the different tables is important. For example, suppose you have a table containing employees' data, such as their name, department, and salary, and you have another table containing department information, such as its name, its manager, and its yearly budget. Suppose you want to answer the question "who's the (department) manager of employee X?". In that case, you will need to look at the first table to find out which department the employee belongs to, and then at the second table to see who is the department manager for that department, provided departments are identified identically in both tables. You can look at two separate tables to obtain a data point, because those two tables are related (in this example, by the department name or ID). In a relational database, a record is often split (or "normalized") and stored in separate tables, and relationships are defined through the use of *primary* and *foreign* keys.

> **Important note**
>
> **Normalization** is a process defined in the relational model for databases that refers to the organization of data. The goal of this organization is to eliminate data redundancy and undesirable inconsistencies when inserting, updating, or deleting records. There are a few rules (referred to as "normal forms") that dictate ways in which to organize data in order to achieve these goals.

So, when you think relational data, think organized tables. Applications handling relational data rely on a **schema** that defines the **data structure**. The schema defines in a formal language how the data is organized and what kinds of data there are. Specifically, it defines the data entities with their precise data types as well as the relationships between them. The schema imposes integrity constraints on the database, which makes relational databases reliable for querying and processing data but somewhat inflexible when it comes to expanding or modifying its structure (schema). **Structured Query Language** (**SQL**) is the standard language associated with relational databases for data management, which is why they're often also referred to as SQL databases. With SQL, you can access and manipulate data in databases. For example, you can execute statements to perform tasks such as inserting or updating data in a database, or retrieving data from it.

On the other hand, non-relational data refers to pretty much any other type of data, for example, a JSON file, an image, or a time series record of a device's temperature reading. These are often (though not always) non-tabular data, where each entity is "self-contained" and doesn't hold data that is related (from a data management perspective) to any other data record. Non-relational databases, which are often also referred to as **NoSQL** databases, are typically more flexible, the reason being they don't require a schema to be enforced, and they can more easily be partitioned, which facilitates **horizontal scalability** and their ability to grow (more on that shortly). Non-relational databases are less complicated and easier to manage in general. This means they are, in a way, more developer-friendly as they can accommodate a complexity of data inputs without having you enforce a schema or structure and without requiring you to run complex *join* and other such (SQL) queries typical in the management of relational databases. IT teams that work with large and complex relational databases will typically have at least one **Database Administrator** (**DBA**) member of staff who is solely responsible for managing the databases and ensuring that they meet capacity and performance requirements at all times.

Teams working with non-relational databases, on the other hand, will typically have an easier time managing them and, in most cases, not require intervention for scaling the databases to meet demand or to fine-tune them for better performance. Part of the reason why they generally perform better is that non-relational databases are optimized for specific data models. They are therefore further categorized based on the particular data model they work with. These categories are as follows:

- **Key-value Store**: For data that is stored as a set of key-value pairs. A record is uniquely identified by its key. As with any of the NoSQL datastores, there is no enforced schema. In addition, in key-value stores, each data entity is treated as a single opaque collection, which may have different fields for different records.
- **Document Store**: For document-oriented information, often referred to as semi-structured data. Document stores are a subclass of key-value stores, the difference being that instead of treating data as opaque objects, a document database can use the underlying structure of the document (a "document" here referring to text-based content with a recognizable hierarchical syntax, such as JSON or XML) to extract metadata that the database engine can use for optimizing data queries and processing. Documents in a document store are somewhat equivalent to the concept of an object in software programming. An object can be an instance of a particular class (with a defined "schema" or set of attributes), but can also have the flexibility to have its own set of attributes and differentiate itself from other objects of the same class. Storing information from an application object into a document is usually very straightforward, as long as that object is serializable into JSON content, for example.

- **Column-oriented Database**: For tabular data where records are stored by columns rather than by row. These types of databases are typically designed to support SQL language for queries and data operations, but they differ from traditional relational databases in that, by storing data in columns rather than rows, it can achieve better performance when finding and filtering through data. In practice, these databases are well-suited for analytics workloads (for example, data warehousing), which typically rely on many complex queries over the entire database. Also, because this increase in query efficiency comes at a slight cost for data insertion, it is not very well suited to transaction-heavy workloads.
- **Graph Database**: For data that contains relationships between the different data points ("nodes") in a graph-like structure. Graph databases are more concerned with the relationship between entities than the entities themselves. The most common use case for this is social networks. A graph database would be well suited to store the relationship between social media members (in other words, who is friends with whom), though not necessarily suited to store information relating to the members themselves (other than their identifiers). Relationships are a first-class citizen in a graph database and can be given several properties (label, weight, direction, and so on). Graph search and other graph operations are optimized for these databases.
- **In-memory**: For data that benefits from memory storage (as opposed to disk storage) for faster access. Memory access is generally significantly faster and with more predictable performance than disk access, which makes query times and overall performance better. This is especially useful for applications that are latency-critical in data retrieval and require fast response times. The price to pay is RAM's volatile nature, which makes this a slightly less reliable form of data storage that comes at a risk of data loss should a power loss event occur. In-memory data systems can be designed to continuously replicate data to separate instances to preserve data durability in the event of the failure of the primary instance.

It's not uncommon in modern cloud design to include not one but a combination of these solutions, so that different types of data are placed in different database services based on what fits them best. This approach of using mixed data storage technologies for varying storage needs is often referred to as **polyglot persistence**.

You should now understand how to identify whether data is relational or non-relational in nature, based on all the concepts you just learned. Next, let's look at a few other ways to categorize data.

Structured versus unstructured

The second distinction to draw is between **structured** and **unstructured** data. Structured data will have an identifiable and repeatable structure for all data points. For example, a table consists of structured data. An HTML file where there is a placeholder for the headers, the body, the paragraphs, and suchlike consists of structured data (or, to be a little more strict with the definition, semi-structured data in this case). So does a JSON file, with a schema that defines what the keys and their value types are. For example, the following screenshot shows what a JSON schema could look like:

```
{
  "title": "Book",
  "type": "object",
  "properties": {
    "title": {
      "type": "string",
      "description": "The book's title"
    },
    "author": {
      "type": "string",
      "description": "The book's author"
    },
    "numOfPages": {
      "description": "The total number of pages in the book",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

```
{
  "title": "The Old Man and the Sea",
  "author": "Ernest Hemingway",
  "numOfPages": 127
}

{
  "title": "The Old Man and the Sea",
  "author": "Ernest Hemingway",
  "numOfPages": "127"
}

{
  "title": "The Old Man and the Sea",
  "author": "Ernest Hemingway",
  "numOfPages": -1
}
```

Figure 5.1 – JSON schema example

On the left-hand side, a schema for a book object is defined. On the right-hand side, three instances of a book are shown: the one on top is a valid book object according to the schema, while the other two have badly defined values for the `numOfPages` property.

In other words, structured or semi-structured data is organized and easier to query and process. In the preceding example, you could confidently query a value for the `numOfPages` property of a book and, for example, do a math operation on it since you know that value is supposed to be an integer. As a data consumer, you know what to expect.

Unstructured data follows no pre-defined structure. Images, videos, and other "binary" contents are unstructured data. Texts and documents are generally unstructured as well. You can't easily query, process, or mine data from unstructured data sources. On the other hand, machine learning applications can often leverage these types of data for predictive analytics.

Transactional versus non-transactional

This next distinction concerns the transactional nature of data. Transactional data means that information is recorded from a transaction, in other words, a sequence of related information exchange that is treated as a data request unit. The most common examples of this are data related to product purchases or financial transactions in general (such as money transfers or payments).

When, for example, a customer buys a product, a few different things happen in the seller's database(s): the product's quantity is decreased in the product catalogue table, the cash money is increased by a certain amount, a purchase event is added to that customer's purchase history table, and so on. In this same example, suppose both buyer and seller are customers of the same bank. In the bank's database, the buyer's balance will decrease by a certain amount, while the seller's balance will increase by the same amount, both operations contained within a single transaction. If those data updates were carried out separately, problems could occur. Imagine that, after updating the balance record for the buyer, the bank's system proceeds to update the balance for the seller, and right at that moment, the database goes down or there's a system failure that prevents it from completing the second part of the transaction, and that operation is lost. Or, even if the application is designed to recover from that, there's an unnecessary risk due to a period of time during which the bank's overall money balance is below what it should be. For banks handling millions of such transactions daily, that could quickly become an issue. That is why systems that deal with transactional records have an "all or nothing" principle, where either the entire transaction is recorded successfully, or it will all fail completely, and the database state will be the same as it was before the attempt.

Tied to the concept of transactions is a set of properties referred to as **ACID**: **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These properties are what guarantee data validity for transactional databases even in the event of system failures (such as power loss and hardware failures).

Atomicity refers to the fact that all statements that compose a transaction are treated as a single "unit," and that they all either fail together or succeed together (the "all or nothing" approach). A database that has the atomicity property will always prevent updates from occurring only partially.

Consistency ensures that a transaction will always bring the database from a valid state to another. A valid state is characterized by having data observing all constraints and integrity (as well as referential integrity through the primary key-foreign key relationship), and so that every database client will read the same data on the same query. We will discuss consistency a little bit more shortly in the context of distributed database systems.

The isolation property refers to the isolation between transactions, in other words, it is the guarantee that the concurrent execution of transactions will still leave the database in the same state as if they had been executed sequentially (in other words, one does not affect the other).

Finally, a database that has durability guarantees that once a transaction has been committed and data has been recorded, it will remain so even in the case of a system failure. This usually simply means that data is recorded in non-volatile storage. When we're talking about transactions on an **ACID-compliant** database, we're therefore talking about a sequence of database operations that satisfies all those properties.

Non-transactional data covers everything else. Updating your social media profile or uploading a video to Youtube are non-transactional requests. If you're playing a mobile game and raising your score, your new score will likely be recorded as a non-transactional event on the game server's database.

Sensitive versus non-sensitive and other data classifications

One last aspect to mention about data is related to data classification. Data is most commonly classified according to confidentiality (in other words, is the data sensitive? Does it contain **personable identifiable information** (**PII**), or a company's intellectual property? Does it contain credit card numbers or private information?). It can also be classified based on whether it's business-critical (used by line-of-business applications) or non-critical data. For companies dealing with compliance obligations, data can be classified, for example, as falling under the protection of the **General Data Protection Regulation** (**GDPR**), the **Health Insurance Portability and Accountability Act** (**HIPAA**), or other such compliance regulations.

Data can also be classified based on its availability (or accessibility) requirements. For example, is the data *hot* and needs to be accessed frequently by applications, or is the data *cold* and can be stored away or archived and only retrieved when needed? All such classifications will impact how the data must be stored and transmitted, as well as its durability and retention requirements, which ultimately help you determine which data storage solution and access tier (when applicable) to adopt.

Now that we've explored the different ways in which we can categorize and classify data, we need to look at one more important bit of theory before exploring storage services in Google Cloud. That is the **CAP theorem**.

The CAP theorem

What really determines the ability to grow and scale in the cloud world (or in any IT system, for that matter) is the capacity to scale *horizontally*. This certainly doesn't apply only to compute systems, but also data systems. However, one crucial thing to understand and keep in mind when working with horizontally scalable, distributed database systems is the CAP theorem. CAP stands for **Consistency**, **Availability**, and **Partition tolerance**, three properties of database systems. The theorem basically states that you can only have two out of the three. In other words, looking at the following diagram, you can only be in one of the shaded areas at the intersections:

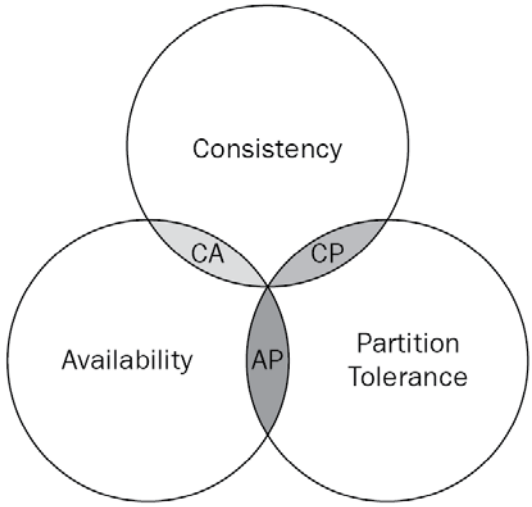

Figure 5.2 – CAP theorem

The partition tolerance property means that the system is tolerant against network partitioning. In other words, it continues to run even in the event of network failures (network nodes or links) that don't result in a general failure of the entire network. For this to work, data records must be sufficiently replicated across combinations of individual database instances to keep everything running without data loss. No network in this world (or any complex system in general) is failure-proof and, therefore, *any distributed database system must be partition tolerant as a design constraint*. This means that the *P* in *CAP* is not an option; it's a must. Something else has to give.

The *Consistency* property means that any *read* operation will return the same value, that of the most recent write operation *across the system*. A system is considered to have consistency if a database transaction starts with the system in a consistent state and ends with the system in a consistent state. During the transaction, there will be a temporary shift into a state of inconsistency. However, as discussed previously, because a transactional database will either apply the entire transaction or rollback entirely if there is an error at any stage, the system will never get "stuck" in an inconsistent state. From this, it can also be concluded that non-transactional databases cannot guarantee consistency, at least not strict, "strong" consistency, as it's often referred to, as opposed to "weak" consistency (usually synonymous with eventual consistency – a specific form of it). To understand the difference between strong and weak consistency, first, we need to understand the *Availability* property (the *A* in *CAP*).

The *Availability* property of distributed databases states that every request from every client will get a response, regardless of the state of any individual server node in the system. If anyone is writing something somewhere to the same database record you're reading and at the same time, the database system will not wait and will not guarantee that you will be reading that latest input (or that you will be reading the same value as someone else reading it closer to the data input node). In other words, to achieve high availability in a partition-tolerant system, strong consistency must be sacrificed.

So, transactional databases typically offer consistency at the expense of availability. This is especially the case for financial institutions, where you cannot possibly afford inconsistency (imagine someone withdrawing money from an account that just went below the required balance due to a transfer operation that happened half a second ago, from another server). Non-transactional databases (or possibly transactional databases as well) that are willing to sacrifice consistency in favor of high availability can do so, as long as the data model (and the business model around that) is such that high availability is really more important than consistency. In fact, in quite a few cases, strong consistency is not required at all, and it's much preferable to have high availability with an eventually consistent system.

Eventual consistency means that the system doesn't meet the consistency property we just discussed ("strong" consistency), but it guarantees that the system as a whole will eventually achieve a state of consistency. The inconsistency window duration will depend on factors such as communication delays and the system load. In other words, it allows itself some time to propagate data inputs across all nodes in the distributed system, knowing that if a few clients are reading slightly outdated records, while other clients are already getting the newest record, is not a big deal. For example, suppose you make an update to your social media profile. In that case, it's not that important that everyone in the world either all collectively see the updated version or the old version. It's fine if it takes a little longer for some readers to see the newest input. This applies to many other types of data as well.

In general, relational databases such as *MySQL* and *PostgreSQL* will aim for strong consistency. Therefore, if you're trying to build a *distributed* relational database system, you will need to sacrifice high availability to some extent. On the other hand, non-relational, NoSQL databases will most often aim for high availability and some will deliver eventual (weak) consistency.

Now that you have the conceptual foundation surrounding data and data types, you should be able to make more informed decisions when designing data solutions. You should now understand how the nature of the data and its specific requirements translate into design constraints that you need to be aware of, particularly regarding what type of databases you must use and the retention and availability configurations to apply.

Let's now explore the data services available on GCP. For each option, we will look into what use cases they would be best suited for, what is offered in terms of availability and backup, as well as things such as data durability and consistency. We will start with relational and structured datastores, and then we're going to discuss non-relational and unstructured options.

Using relational and structured datastores

In Google Cloud, there are two main options for managed relational database services: **Cloud SQL** and **Cloud Spanner**. From what you've learned in the previous section, you should be able to make an informed decision about when to consider a cloud relational database. In short, they would be well suited for compatibility with existing relational data (for example, if you're migrating from an on-premises **MySQL**, **PostgreSQL**, or **SQL server**), or if you're dealing with transactional data or data that needs to preserve a relational structure over which complex queries and JOIN statements are expected. We will explore the capabilities of the managed relational databases on GCP and when you would choose one over the other.

Cloud SQL

Cloud SQL is a fully managed database hosting service for *Microsoft SQL Server*, *MySQL*, or *PostgreSQL*. The platform provides security at rest and in transit by default, with customer data encrypted on Google's internal networks and in database tables, temporary files, and backups. For increased availability, you can configure replication and enable automatic failover. Data is backed up automatically every day, and you can also configure a backup policy for the data you store in the database, as well as **point-in-time recovery** settings. Cloud SQL is SSAE 16-, ISO 27001-, and PCI DSS-compliant and supports HIPAA compliance.

High availability (HA) and failover

For a high-availability configuration, you can deploy Cloud SQL as a regional instance. It will then be located in a primary zone within the configured region and replicated to a secondary zone (as a **Standby** instance) within the same region. The replication is set up as **Synchronous replication**, so that all writes made to the primary instance are also immediately applied to the standby instance. This setup is illustrated in the following diagram:

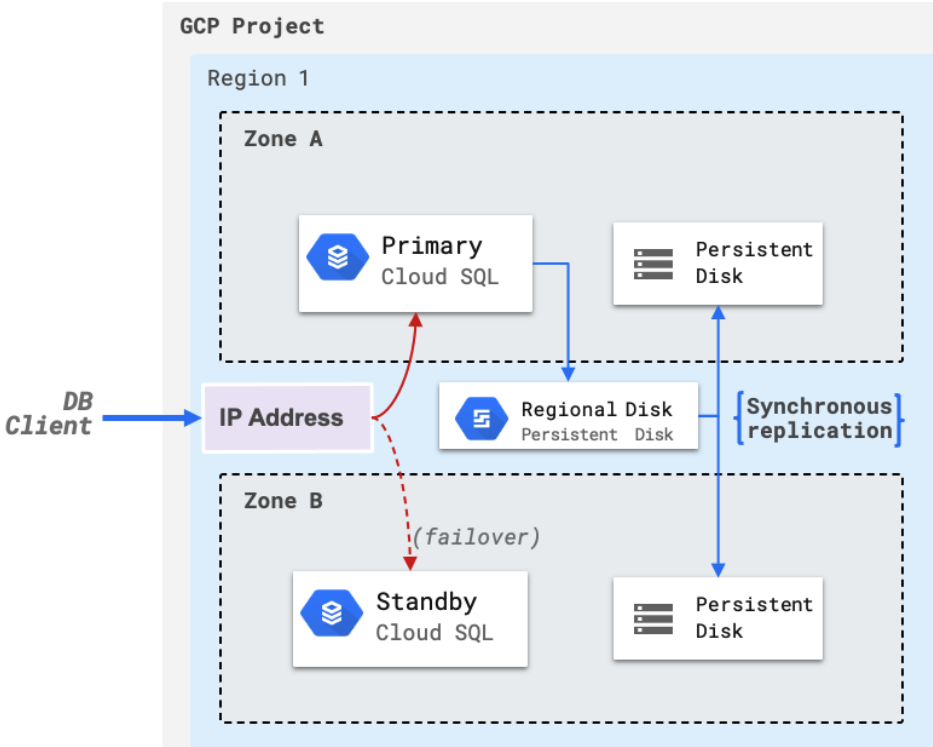

Figure 5.3 – Cloud SQL HA setup

If the primary instance or the entire zone becomes unresponsive (for approximately 60 seconds), the platform will automatically switch to serving data from the **Standby** instance (**automatic failover**). Thanks to a shared static IP address, the **Standby** instance will serve data behind the same IP from the secondary zone, which means connection strings won't need to be updated from client applications. Although the *failover* is automatic, if the **Primary** instance (or zone) recovers, there will not be an automatic *failback*: you will need to perform it manually.

In addition to leveraging backup and point-in-time restore capabilities, consider designing a Cloud SQL solution with the high-availability setup and automatic failover enabled, certainly in production environments. It will save you time, effort, and considerable downtime in the event of a zonal or instance failure, not to mention the fact that it reduces the risk of data loss in case of a zonal disaster that would cause the zone to be deemed unrecoverable.

Replication

Replication of a SQL database can serve a few different purposes. For example, it can be used for scaling out read operations (and offloading the main server), migrating data between regions or even platforms, and also for increased availability and durability in case data in the original instance becomes corrupted or inaccessible. Replicas can only be used for *read* operations (not *writes*) and are therefore referred to as **read replicas**. They process queries, read requests, and analytics workloads.

In Cloud SQL, you can create an **in-region read replica** (where a read replica is created in the same region, but different zone) or a **cross-region read replica** (where a read replica is created in a separate region) for MySQL and PostgreSQL databases. With MySQL, you can also configure replication as an **external read replica**, when the database instance is replicated to another that is hosted by another cloud platform or on-premises environments. Although not "natively" supported, you would also be able to set up Microsoft SQL Server replication using Microsoft SQL management tools.

Consider creating a read replica if the main server is at heavy load. You can then point some of the clients (in particular, the ones that perform analytics and heavy read operations) to the replica (which has its own connection strings). This is essentially a read scale-out operation. Other reasons to consider it include the following:

- Reducing latency for clients in different regions (consider the cross-region read replica)
- Kicking off the migration of data to another region or platform
- Increase overall read capacity in anticipation of heavy read events

Backups and point-in-time recovery

Backups protect your data from unexpected loss or corruption. With Cloud SQL, you can enable automated backups, which, along with binary logging, is a requirement for some operations (such as clone and replica creation). Cloud SQL retains up to 7 backups for an instance, but the backup storage is priced at a lower rate than regular storage. In addition, only the oldest stored backup will be as large as your database instance, since the first backup is a full backup while subsequent backups are incremental. An incremental backup includes only data that changed after the previous backup, not the full dataset. For example, if a database holds a collection of employee records with 500 employees, a full backup is taken to include all 500 employees. If 2 new employees were added to the table since the full backup was taken, a subsequent, incremental backup will hold only those 2 new employees. Should you need to restore all data at this point, your SQL server would apply the full backup first, followed by the incremental backups, meaning that you would have a total of 502 records. If there were multiple incremental backups, they would have been applied sequentially. When the oldest backup is removed (which was your only full backup), the next oldest backup becomes a full backup by taking in all the data.

Backups can be performed on demand or in an automated way. An on-demand backup can be created at any time (regardless of whether the instance has automated backups). For example, if you're about to make a risky change to your system (this may not necessarily be directly on the database itself, but something that may impact it), it's a good practice to create a backup. On-demand backups will persist until you manually delete them (or until the instance is deleted), which is something to keep in mind from a cost management perspective. Manual backups lingering around can quickly become a source of unnecessary high storage costs. When enabled, automated backups happen every day during a customizable 4-hour window, at the start of which the backup operation will begin. When possible, you should make sure that the window fits into a window of least activity on your database to prevent performance degradation from occurring at a moment of high usage.

By default, Cloud SQL stores backup data in two regions (chosen automatically by the platform) for redundancy and resiliency against regional failure. One region can be the same as the region the instance is in. The other is guaranteed to be a different region. You can, however, also define custom backup locations. This is useful if there are compliance and data residency regulations you need to observe in the organization you're working with. For example, if data needs to be kept within a geographic boundary, you will need to customize the backup locations to satisfy that requirement.

Related to backups, *Point-in-time recovery* is a feature that is enabled by default with MySQL and PostgreSQL options and which allows you to recover a database instance back to the state it was at a specific point in time. This is useful when an error causes a loss or corruption of data, in which case you can just restore the database to a moment right before when the error occurred. A point-in-time recovery always creates a new database instance (it's not an in-place restore), similarly to a clone operation, so that instance settings are inherited. If you're running Microsoft SQL Server, this is not a "native" platform feature in Cloud SQL. Instead, you need to set up backups for point-in-time recovery yourself using Microsoft **SQL Server Management Studio** (**SSMS**) (and, for example, uploading them to a storage account). Specific guidance on how to set this up can be found at `https://cloud.google.com/solutions/backup-and-archival-of-sql-with-point-in-time-recovery`.

Maintenance

There is no server for you to manage when you provision a Cloud SQL service. However, there is still a server for Google Cloud to operate, and, for that reason, you need to be aware of maintenance events. Instances will need occasional updates for patching, fixing bugs, and performing general upgrades. All these events generally require a restart of the instance, which can disrupt your service (maintenance events do not trigger a failover to standby instances). In Google Cloud, you can set your preferred maintenance windows and options on instances. You can control, for example, the day of the week and time when an instance should receive updates if there are any. If you don't specify a preferred window, disruptive updates can happen at any time. Even if you do specify a preferred window, however, some high-priority maintenance events (such as critical service updates or patches for vulnerabilities) will likely be rolled out without waiting for your specified window. These disruptions count as downtime against the platform's SLA, but to prevent serious disruptions to your applications, there are some general design recommendations that you can apply to applications communicating with the databases. These include *connection pooling*, *short-lived connections*, and *exponential back-off for connection retries*. More details and guidance for specific programming languages and SQL engines can be found at `https://cloud.google.com/sql/docs/mysql/manage-connections`. This is an excellent resource to have bookmarked and shared with the development team when you're designing a Cloud SQL solution for them.

Cloud SQL Proxy

The Cloud SQL Proxy provides secure access to Cloud SQL database instances without the need to configure SSL and or network whitelisting. Database clients can access instances via the Cloud SQL Proxy with secure, automatically encrypted traffic (using TLS 1.2) and managed SSL certificates. The way Cloud SQL Proxy works is illustrated in the following diagram:

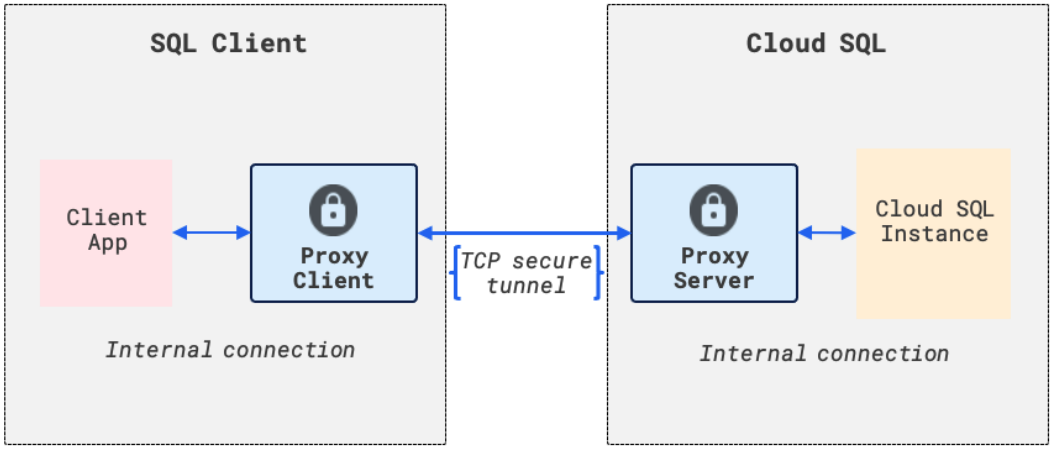

Figure 5.4 – Cloud SQL Proxy architecture

The proxy handles authentication with Cloud SQL and network connectivity without the need for you to set up static IP addresses (the instance still needs to have either a public IPv4 address or a private IP). The applications then communicate with the proxy using the standard database protocol used by the database (in other words, the proxy "impersonates" the database). This communication is represented by the bi-directional arrow between **Client App** and **Proxy Client** in the preceding diagram. The proxy itself can authenticate against the database with a service account associated with a credential file, permanently linked to the proxy as long as it is running.

Access control

Access control can be configured at the instance level and at the database level. The instance-level access configuration method depends on the connection source, but generally, it involves a Cloud SQL Proxy that is either set up by you or automatically by a serverless application service (for example, App Engine, Cloud Functions, or Cloud Run). Once a connection to an instance has been established, the user or application can log in to specific databases with a user account that has been created as part of managing your Cloud SQL instance. The default user (root) must always be set up when creating a new instance, but additional users with fine-grained permissions can be created.

We will discuss audit logs and other security considerations in *Chapter 7*, *Designing for Security and Compliance*. Let's now look at one final aspect within Cloud SQL that can't be overlooked by a cloud architect: *scalability*.

Scalability

As we've discussed a few times in this book, there are two ways in which you can scale a service: scaling up (increasing the capacity of the current serving instance or platform), also referred to as vertical scaling; and scaling out (increasing the number of serving instances), also referred to as horizontal scaling. Scaling up is the easiest operation as it doesn't impose any requirements on the applications running inside the instance, but it's the option with which you will hit a limit much quicker – after all, you can only increase the capacity of any instance up to the available capacity of the underlying hardware (and the available capacity to *you*, specifically, since we're talking about a shared, multi-tenant infrastructure). In addition, a performance bottleneck may arise not from the lack of compute, memory, or storage power, but from the application's inability to handle any more requests. This is not to mention the fact that an instance may fail in many ways and require a restart or a downtime period. For all those reasons, being able to *scale out* is a crucial goal that we should strive to achieve when architecting cloud infrastructure.

If you need to scale up your Cloud SQL instance, that is easy enough to achieve. Update its machine type to a larger type. This will increase memory and the number of virtual cores, as well as the storage and IOPS capacities of the instance as a whole. As of the time of writing, the largest database-optimized machine type for Cloud SQL on GCP comes with 96 vCPUs and roughly 368 GB of memory. This is a powerful machine indeed, but should you need to scale out your database, let's look at how we can achieve this with Cloud SQL.

We have already mentioned the strategy of using read replicas to scale out read operations for Cloud SQL instances. This is an easy way to scale out, since there's still a single "source of truth," so to speak, or a single database where all the writes happen (and therefore all and any modifications to the existing data are performed at one location). This removes concurrency concerns (concurrent *read* operations are much less of a concern) and is a more or less a frictionless way to scale out. It comes, however, with the obvious limitation of not allowing for the scaling out of *write* operations. For reasons you're now already familiar with from the previous section (remember the CAP theorem in particular), it is not as straightforward to scale out relational databases as it is with non-relational databases, at least not in a way that preserves ACID properties and strong consistency.

There is one strategy, however, that accomplishes a level of horizontal scaling relatively well while somewhat preserving ACID properties. That is **sharding**. Sharding involves partitioning the database into smaller parts (shards, or partitions), and then distributing the shards across a cluster of machines. However, it usually does so at the expense of no ACID guarantees or referential integrity when performing queries and transactions across shards. Data queries are automatically routed to the corresponding server based on either application logic or a query router. One common and simple example of how you can apply this is on a database of users: you can create a shard for each letter of the alphabet (or a grouping of letters, such as *A-E*, if you don't need that many shards). That way, when running queries against the database, you know based on the first letter of the user's name from which particular shard that should be retrieved (or written to).

Sharding is performed particularly well with **MySQL Cluster**, a feature in MySQL that automatically shards tables across nodes. It does so in order to not lose the ability to perform "join" operations (cross-shard queries) without sacrificing ACID guarantees. However, this is a complex problem, and some argue that there can't ever be true horizontal scalability with transactional databases. But we will see shortly that we still can do it if we're willing to make just a little compromise.

You have now learned the most fundamental tenets of a well-designed relational database solution using Cloud SQL. We looked into the high-availability setup you should aim for and features such as backup, point-in-time recovery, and automatic failover, which you can leverage with this service. We then discussed the use of replication and things you need to be aware of in terms of maintenance and access control. Finally, we addressed the elephant in the room of relational databases: horizontal scalability, and how it can become a very complex, borderline impossible problem. Google challenges this notion with its managed relational database service, which we're going to look into next: Cloud Spanner.

Cloud Spanner

Cloud Spanner is Google Cloud's fully managed and *horizontally scalable* relational database service, with *strong consistency* and up to 99.999% availability. This service was designed from the ground up to deliver ACID-compliant, high-performance transactions globally with automatic handling of replicas, sharding, and transaction processing. It's a service built initially for Google's own critical business applications, but made available on GCP.

In a nutshell, the way Cloud Spanner works is by partitioning the database tables into contiguous key ranges called *splits*, with a fast lookup service for determining the machine(s) that serve a key range. Splits are replicated to multiple machines in distinct failure domains to prevent data from becoming inaccessible due to server failures. This is similar to an automatic sharding mechanism that ensures availability. Consistent replication is managed by a **Paxos algorithm**, which implements a voting system to elect a *leader* among replicas to process writes, while all other replicas serve reads.

> **Important note**
>
> **Paxos** is a family of protocols created to solve consensus in unreliable networks (which real networks are), and the use of a Paxos algorithm is a core design choice in Cloud Spanner. If you want a more in-depth look at the problem of consensus in distributed systems and how Paxos is applied, check out this chapter from the Google SRE book: `https://sre.google/sre-book/managing-critical-state/`.

Read transactions will always read the latest copy of data, and write transactions introduce locking on the database in a way that is orchestrated by *Paxos leaders* and involve (very complex) cross-machine communications. With a Google-developed technology called *TrueTime*, every machine in Google data centers knows the exact global time with a high degree of accuracy – this is what allows different Spanner machines to reason about the ordering of transactional operations, and this is one of the things that make Spanner possible. Most importantly, however, all of the details just mentioned are transparent to Spanner clients. The protocols and technologies to ensure data consistency despite the globally distributed nature of the database are much more complicated than that and, if you're curious to know (or a skeptic about the whole thing), they're described in a research paper available at `https://research.google/pubs/pub45855/`.

If you're wondering about the CAP theorem and whether Cloud Spanner "breaks" it, it doesn't, at least not from a technical, purist perspective, but indeed it effectively does (in other words, the users *perceive* it as a CAP system). What is happening is that, in the event of network partitions, Spanner chooses consistency and forfeits availability. Strictly speaking, it's a CP system. However, if its actual availability is so high (given the well-engineered redundancy in the solution) that users can ignore outages, then Spanner can justify an effectively CAP claim. After all, the fact that a database system is not 100% but 99.999% available (one failure in 100,000) is barely noticeable when the applications themselves (and other infrastructure components) have even lower availability SLAs.

From the clients' perspective, Cloud Spanner works in much the same way as any other SQL-based relational database. When executing transactions, however, there are a few things to keep in mind to ensure that the system delivers on its consistency promises:

- If you need to write data to the database, depending on the value of one or more reads, then you should execute the read as part of a **read-write** transaction.
- If you need to make multiple read calls that require a consistent view of your data, those should be executed as **read-only transactions**.

Cloud Spanner differentiates transactions by one of three types it supports: a *locking read-write transaction*, which is the only type that supports writing data; a *read-only transaction*, which provides guaranteed consistency across several reads; and partitioned **Data Manipulation Language** (**DML**), which is designed for bulk updates and deletes. Even though read-write transactions can also read data, you should only use this type of transaction for reading when actually making a subsequent write as part of the same atomic transaction (or even if you just *might* do a transaction depending on a read's value, this can also be used).

When it comes to reading data, Cloud Spanner offers two types of reads: a *strong read*, in which data is read at a current timestamp and is guaranteed to see all data that has been committed up until the start of the read; and a *stale read*, in which data is read at a timestamp in the past.

You may want to consider bookmarking the following references to share with developers and database administrators regarding reads, transactions, and best practices for constructing SQL statements that will help Cloud Spanner optimize execution performance:

- `https://cloud.google.com/spanner/docs/reads`
- `https://cloud.google.com/spanner/docs/transactions`
- `https://cloud.google.com/spanner/docs/sql-best-practices`

In Cloud Spanner, backups are not created automatically but on demand, and they can be retained for up to one year. Longer retention times can be achieved by exporting the database (to a storage account, for example), and if you need periodic backups, you can certainly automate the process.

To get you familiarized with Cloud Spanner, we will create an instance in two geographic regions, create a database, run some simple SQL queries using **gcloud**, and then create a backup. It may sound like only a simplistic "hello-world" type of deployment (and indeed it is), but it is effectively delivering a *multi-regional, scalable, and highly available relational database service* from only these steps – about all you need when you're designing a production relational database solution for organizations, minus the IAM policies, audit logs, as well as other security and compliance components, which we will get back to in a future chapter.

Creating a scalable and highly available SQL database with Cloud Spanner

Let's follow these steps to create the database:

1. Go to the GCP console (`console.cloud.google.com`) and then click on the shell icon in the top-right corner of the screen to activate Cloud Shell:

Figure 5.5 – Activating Cloud Shell

2. In the Cloud Shell terminal, run the following command:

```
$ gcloud projects create --name chapter-5 --set-as-
default
```

3. In the console, click on the down arrow next to where it says **Select a project** (or your current project's name) to open the project selection menu. Click on the **All** tab to see all projects, and then click on the newly created project (**chapter-5**), as shown in the following screenshot:

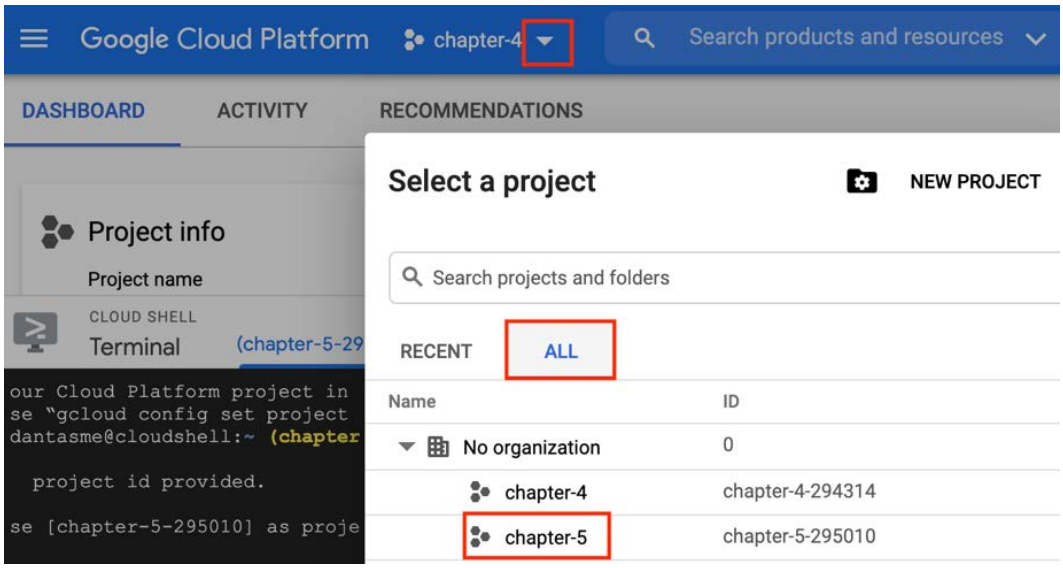

Figure 5.6 – Project selection menu in the console

4. Still in the console, click on the stacked lines icon in the top-left corner of the screen to expand the left-side menu (if not already expanded), as shown on the left-hand side of the next screenshot, and then search for and click on **Spanner**. You will be prompted to enable the Cloud Spanner API. Do so by clicking on **Enable**, as shown on the right-hand side of the following screenshot:

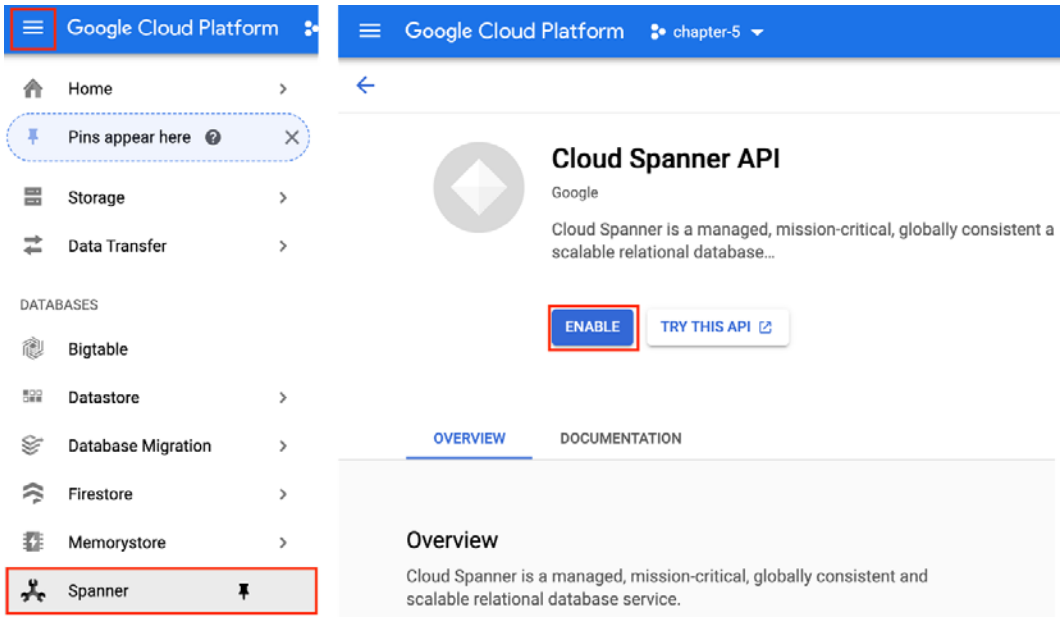

Figure 5.7 – Enabling Cloud Spanner API

5. You will be prompted to enable billing on your project. Click on **Enable Billing**, and then, in the billing account selection menu, expand the **Billing account** drop-down field and click on your existing billing account. Finally, click on **Set Account**.
6. On the **Spanner** page, click on **Create Instance**. Fill in the creation form as follows:
   a) **Instance name**: `spanner-instance`
   b) **Choose a configuration**: **Multi-region**
   c) **Select a configuration**: Choose either one of the available multi-region configurations, for example, **eur3 (Belgium/Netherlands)**.
   d) **Nodes**: `1`
7. Click on **Create**. Next, click on **Create Database**.
8. In the database creation form, type `testdb` in the **Name** field, and then click on **Continue**. In the **Define your database schema** field, make sure that the **Edit as text** option is selected and paste in the following under **DDL statements**:

```
CREATE TABLE Customers (
  CustomerId   INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024),
```

```
    BirthDate  DATE,
  ) PRIMARY KEY(CustomerId);
```

9. In the left pane of the console, click on **Customers** under the database's name (**testdb**). Next, click on the **Data** tab. Finally, click on **Insert**. This is a way to insert data easily through a graphical user interface. Just add two rows to this table by entering some fake customer information (making sure **CustomerId** is unique), as shown in the following screenshot, for example:

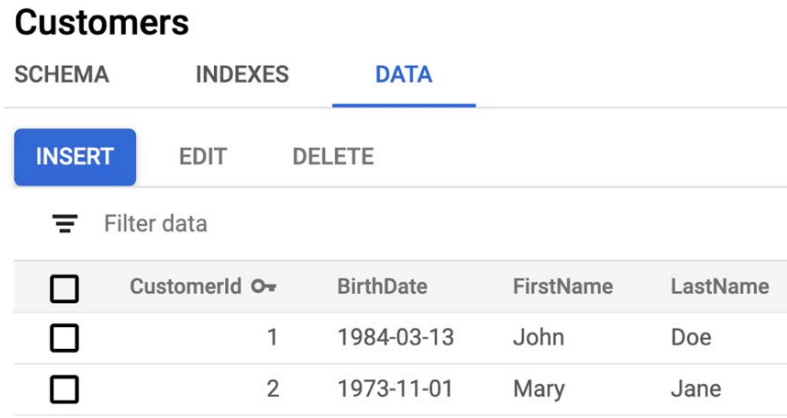

Figure 5.8 – Customers' table

10. Now, click on **Query** in the left pane, under the database's name (**testdb**). Then, paste the following into the textbox:

```
  SELECT * FROM Customers
```

You should see the two users you created in the **Results table** section at the bottom of the page.

11. Finally, click on the name of the instance (**spanner-instance**) on the left. Next, click on **Backup/Restore**. Under **Backups**, click on **Create**. Then, fill in the form as follows:

    a) **Database name**: `testdb`

    b) **Backup name**: `testdb-backup1`

    c) **Set an expiration date**: **1 day**

12. Click on **Create**. You will be taken back to the instance details page. Scroll down until you see **Backup operations** and notice the on-going backup operation.

To ensure external consistency of the backup, Cloud Spanner is now pinning the contents of the database to prevent the garbage collection system from removing relevant data values for the duration of the backup operation. A backup operation happens in parallel on the different zones the instance is replicated to. The database is restorable as soon as the backup operation has been completed.

13. As a final step, go to Cloud Shell (if you don't still have it open, open a new one, making sure you've selected the right project by running `gcloud config set project` and pasting in your project ID). To illustrate how you can run a SQL query using `gcloud`, run the following in the terminal:

```
$ gcloud spanner databases execute-sql testdb
--instance=spanner-instance --sql='SELECT * FROM
Customers WHERE CustomerId = 1'
```

14. You should see the *John Doe* customer as a result (make sure you modify the `CustomerId` value if you've set your own values).

    Cloud Spanner has several client libraries for different languages and frameworks. If you wish to continue exploring it with your preferred client, check out the various available libraries and usage instructions at `https://cloud.google.com/spanner/docs/reference/libraries`.

15. To remove the database and clean up all resources, first, we need to remove the backup. Back in the console and on the Spanner page, first click on the instance name (**spanner-instance**) and then click on **Backup/Restore**. Select the backup, **testdb-backup1**, and then click on **Delete**. When prompted, confirm the deletion by typing in the backup's ID (`testdb-backup1`), and then click on **Delete**. Finally, still in the instance details page, click on **Delete Instance**. Once again, confirm by typing in the instance's ID (`spanner-instance`) and then clicking on **Delete**.

You have now learned and explored the relational database options in Google Cloud, their features and capabilities, and how to design them for availability and scalability. Also, you now know what situations they are well-suited for. Generally speaking, if you don't have any of the requirements that call for relational databases (such as ACID compliance and schema enforcement), then a non-relational database is what you should aim for. They're fast, easily scalable, and developers will usually find them easier to work with. We will now explore what the non-relational database options are on GCP.

Using non-relational and unstructured datastores

In general, non-relational data falls under the category of semi-structured or unstructured data. This means that they don't follow any strict tabular structure of data models (as is the case for relational databases), but, in the case of semi-structured data, they do contain elements that enforce a hierarchy of records and fields within the data. The most notable example is **JSON**, short for **JavaScript Object Notation**, which uses attribute-values pairs to describe objects and in which values can potentially be another JSON block, thus allowing for a nested data structure that enables data models to represent complex relationships between entities. The way objects (or entities) are described in JSON is akin to how objects are described in object-oriented programming languages. In addition, the support for lists of objects simplifies data models by avoiding complex translations from lists to a corresponding object in a relational data model.

Non-relational databases have their critics, too. Many argue that the lack of a standard query language such as SQL, and the lack of restraints and schema enforcement, make these types of databases prone to "garbage in, garbage out." That is, they can wind up becoming a dumping ground for inconsistently defined data, making data analytics a difficult, if not downright impossible, task. Furthermore, some argue that even though there are no schemas being enforced, there's still an "implied" schema, in other words, all database clients still need to know what the data is about, how it is structured, and what data types the values are in order to efficiently interact with the data anyway. This brings up the question as to whether they really deliver flexibility, and whether the little flexibility you do get is worth the potential nightmare of running analytics workloads on these systems. Opinionated discussions aside, it all comes down to the fact that applications must be well designed and their data models well architected for non-relational databases to have real value. With best practices and a shared understanding in place, non-relational databases will deliver on their promises.

So, what does GCP have to offer? Let's start big with Cloud Bigtable.

Cloud Bigtable

Cloud Bigtable is a fully managed, scalable NoSQL database service on GCP. It is a *key-value store with high throughput at low latency* and is especially geared toward *large analytical and operational workloads*. It is also well suited for machine learning applications, with a storage engine designed with them in mind. Data is automatically replicated with *eventual consistency,* and the service comes with 99.99% availability (when multi-cluster routing is set up, otherwise 99.9% for single-cluster instances).

It is also an ideal source for **MapReduce** operations, a programming model used for distributed computing, which you will learn more about in *Chapter 8*, *Approaching Big Data and Data Pipelines*. Cloud Bigtable integrates easily with the existing *Apache* ecosystem of open source big data software. For teams that already work with *Apache HBase*, Cloud Bigtable will feel familiar and will offer a few key advantages over a self-managed HBase setup, namely:

- **Scale**: Self-managed Hbase installations have a design bottleneck that limits the system's performance after a certain threshold is reached. Cloud Bigtable does not have this bottleneck and can scale in proportion to the number of machines in the cluster.
- **Simplicity**: Cloud Bigtable handles aspects of the underlying platform (such as upgrades, restarts, and replication) transparently. In addition, it automatically maintains high data durability.
- **Cluster resizing without downtime**: You can increase a cluster's size (or reduce it) without any downtime. Scaling events won't affect running workloads.

Ideal use cases for Cloud Bigtable include **ad tech** (for marketing data, such as purchase histories and customer preferences), **fintech** (for financial data, such as transactions, stock prices, and exchange rates), **social/digital media** (for graph data, such as information about relationships between users), and **Internet of Things** (**IoT**) (for time series or IoT data, such as CPU usage over time or telemetry from sensors and appliances). Generally, these applications need high throughput and scalability for reads and writes of key-value data (with values not typically being of a large size). Cloud Bigtable also excels in MapReduce operations and machine learning applications. In fact, Cloud Bigtable powers many of Google's own core services, such as *Search* and *Maps*, and is, therefore, battle-tested for scale and heavy analytics.

So how is data stored in Cloud Bigtable? This database uses a proprietary storage system built on a few Google technologies, such as *Google File System*, *Chubby Lock Service*, and *SSTable*. Data is stored in tables, each of which is a sorted key-value map. Each row of the table defines a single entity and is indexed by a single row key, while the columns contain the values for each row. So far, not very different from a tabular, relational database. The main differentiating characteristic is that columns that are related to one another are typically grouped together into a column family, and each row/column intersection can contain multiple cells (or data versions) at different timestamps, providing a record history of how the stored data has been altered over time. Unlike a relational database, the names and data types of the columns can vary from row to row. In addition, tables are sparse; in other words, if a cell does not contain any data, it does not take up any storage space. Because of these characteristics, Cloud Bigtable falls under the category of column-oriented stores, or, to be a little more accurate, **wide-column stores**. Another such database is *Apache Cassandra*.

With Cloud Bigtable, you can create backups for tables' schemas and data, and then eventually restore them to a new table.

Understanding and setting up instances, clusters, and nodes

To use Cloud Bigtable for your storage needs, you first create an *instance*, which is the container for your data. An instance contains one or more (up to four) *clusters*, spread across different zones. Applications will eventually connect to clusters, each of which comprises nodes (or at least one node), which are units of compute that will perform data management and maintenance tasks. When you create a table, you create it on an instance, which is where the table belongs, and not on a specific cluster or node. If you have more than one cluster within the instance, then tables are going to be replicated.

When you create an instance, there are a few properties that need to be defined, most importantly:

- **The storage type**: This determines whether data will be stored on **solid-state drives** (**SSDs**) or **hard disk drives** (**HDDs**).
- **Application profile**: Especially important when setting up more than one cluster, an application profile (or app profile) will store settings that determine how incoming requests from an application should be handled. This includes routing policy (single- or multi-cluster routing) and whether single-row transactions are allowed.

After creating an instance, you can then create one or up to four clusters, which will belong to the instance. Each cluster must be located in a single and different zone from other clusters. The different zones you choose don't need to be within the same region. With a single cluster setup, your instance will not use replication. If you add a second cluster to the instance, *replication will automatically start* with separate copies of your data placed in each of the clusters and updates being synchronized between copies. You can then either choose to connect different applications to different clusters or have Bigtable balance traffic between clusters. If a cluster becomes unavailable, you can fail over to another, which means that a multi-cluster setup provides you with extra availability.

Finally, each cluster in the instance will be assigned one or more nodes, which are compute resources used by Bigtable. The number of nodes dictates the cluster's capacity to handle income requests and store data, and therefore you must think of monitoring your clusters' CPU and disk usage to add nodes when the metrics exceed a certain threshold. Google Cloud offers the following recommendations on what CPU utilization thresholds to use:

Configuration	Recommended maximum values
Single cluster	70% average CPU utilization 90% CPU utilization of the hottest node
Any number of clusters with single-cluster routing	70% average CPU utilization 90% CPU utilization of hottest node
2 clusters with multi-cluster routing	35% average CPU utilization 45% CPU utilization of hottest node
3 or more clusters with multi-cluster routing	Depends on your configuration

For storage utilization, the general recommendation is to not exceed 70% of the storage capacity.

It is now time to get hands-on and go through a quick example that will illustrate how to create an instance, a cluster, and nodes, and how to write some data to the database that uses a column family structure. For this, we will use the **cbt** command-line tool, which is designed specifically for interacting with Cloud Bigtable. It's possible to use **gcloud** for that (or the console, or any of the available APIs, for that matter), but this tool really simplifies the interactions with Bigtable. This tool is available natively within Cloud Shell, so let's go ahead and open a Cloud Shell session from within the console. Refer to *steps 1 and 2* under the *Cloud Spanner* section header, the *Creating a scalable and highly available SQL database with Cloud Spanner* sub-section, for instructions on how to open Cloud Shell and create a new project (if you haven't already done so):

1. Run the following command in Cloud Shell:

```
$ cbt createinstance cbt-instance1 Instance1 cbt-cluster1
us-east1-d 2 SSD
```

2. You may be asked to authorize Cloud Shell to use your credentials to make a GCP API call. Click on **Authorize** if that happens.

   You will notice that this command will run very quickly. The instance, cluster, and two nodes you've just created are already being spun up in the background, and they'll be ready quite soon. The command has a few positional arguments according to the following syntax:

```
cbt createinstance INSTANCE_ID DISPLAY_NAME CLUSTER_ID
CLUSTER_ZONE CLUSTER_NUM_NODES CLUSTER_STORAGE_TYPE
```

   With that command, we have therefore created one instance with the ID of `cbt-instance1`, a display name of `Instance1`, then, a cluster with the ID `cbt-cluster1`, located in zone `us-east1-d`, with `2` nodes containing `SSD` storage.

3. To add a new cluster and enable replication, run the following command next:

```
$ cbt -instance=cbt-instance1 createcluster cbt-cluster2
us-west1-a 1 SSD
```

   We now have a highly scalable, highly available, wide-column NoSQL database ready to use. In the console, if you expand the left-side menu and search for and click on **Bigtable**, you should be able to see the following:

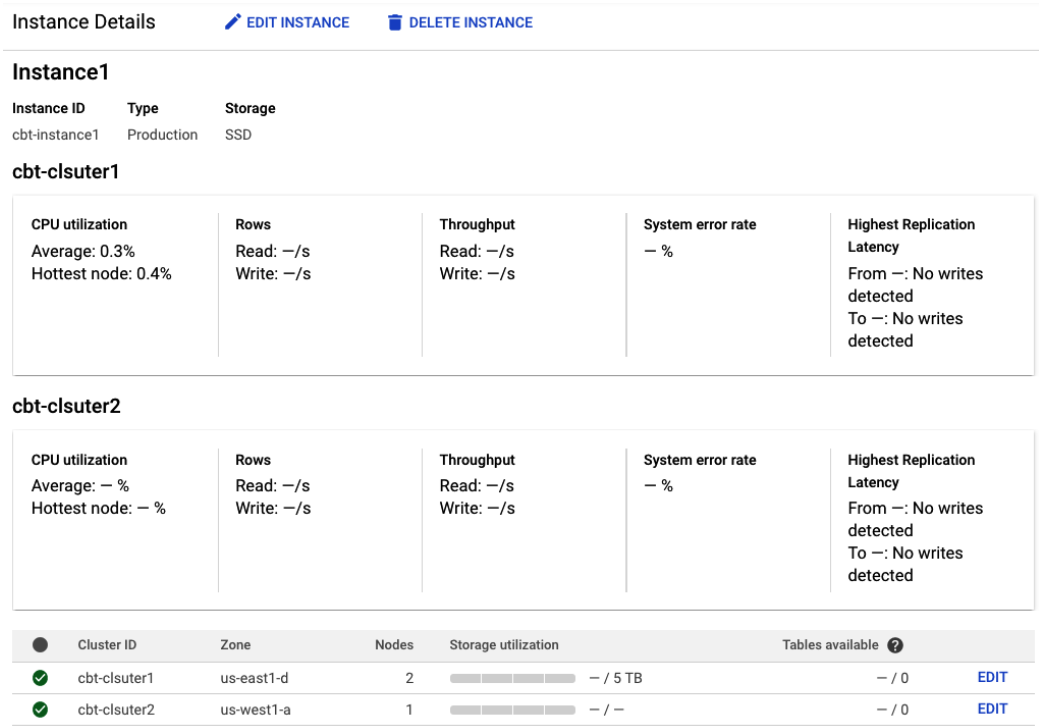

Figure 5.9 – Bigtable Instance Details page

4. Still in the **Bigtable** page, you can click on **Application profiles** to see what the app profile is for this instance. Because we didn't specify any, we should have the default profile, which applies *single-cluster routing* and enables *single-row transactions*.

5. Let's now create a table. Back in Cloud Shell, run the following command to set the default project ID and instance name for **cbt** (replace `PROJECT_ID` with your unique project ID, remembering it's not the same as the project name – **chapter-5**):

```
$ echo -e "project = [PROJECT_ID]\ninstance =
cbt-instance1" > ~/.cbtrc
```

6. Then, run the following two commands to create the table and a column family, which we will call `sensors_summary`:

```
$ cbt createtable iot-table
$ cbt createfamily iot-table sensors_summary
```

7. After running the preceding command, you can go back to the console and navigate to **Tables** from within the Bigtable page. You should see the following page:

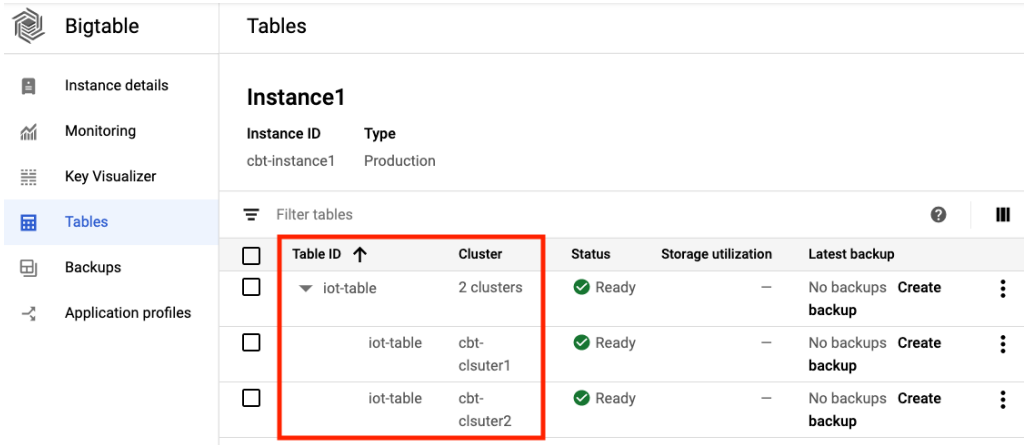

Figure 5.10 – Bigtable table in multi-cluster setup

Note how the table was automatically replicated across the two clusters. Next, let's insert some data. We will do this using the Python client library.

8. Back in the Cloud Shell, download the repository for this book and navigate to the **chapter 5** files by running the following command:

```
$ git clone https://github.com/PacktPublishing/
Architecting-Google-Cloud-Solutions
$ cd Architecting-Google-Cloud-Solutions/ch5
```

9. In the Cloud Shell window, click on the **Open Editor** button located on the top bar:

Figure 5.11 – Cloud Shell Editor

10. In the editor, expand the repository directory, then the `ch5` directory, and the open the `cbt_ingest.py` file. Scroll down in the text editor pane until you see the line that contains `PROJECT_ID`. Replace that with your own project ID. Do the same for the `cbt_read.py` file, which also has a line in the end with `PROJECT_ID`, which you need to edit:

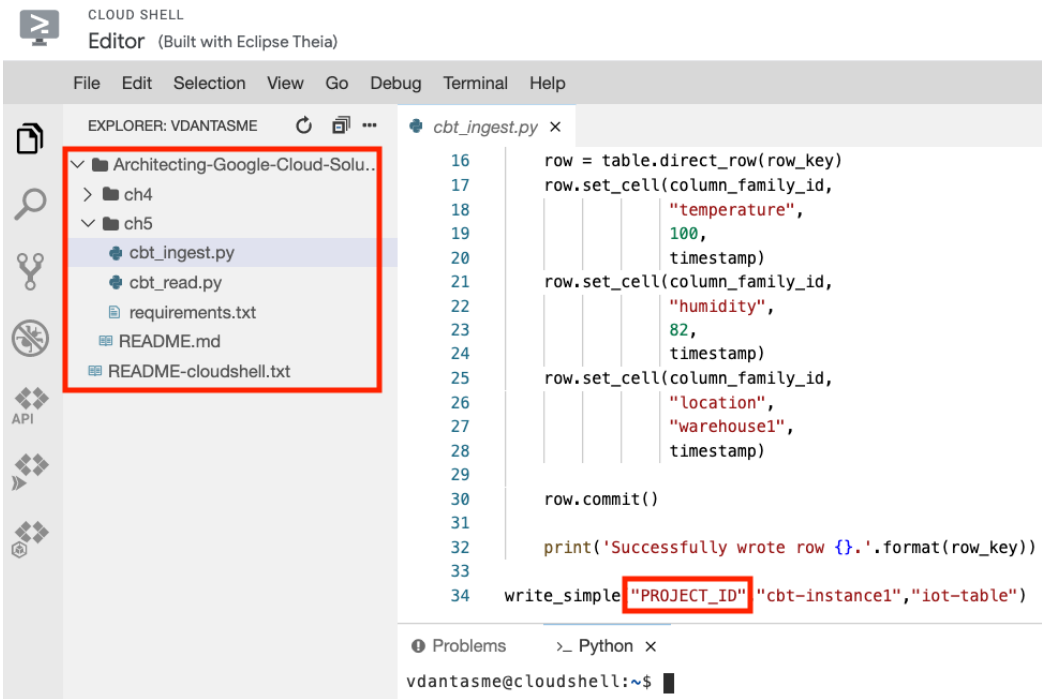

Figure 5.12 – Editing code in Cloud Shell Editor

11. In the Python terminal window on the bottom, run the following command to install the Python Cloud Bigtable library:

```
$ pip3 install google-cloud-bigtable
```

12. Now, go back to the `cbt_ingest.py` file. Click on the *run* icon in the top-right corner of the editor to run this file. You should see an output text in the terminal window below that says, *Successfully wrote row iotdevice#10401*. Next, open the `cbt_read.py` file and then click on the *run* icon. This time, you should see the row you just wrote, which contains some fake IoT device information (temperature, humidity, and location). A timestamp of the record will also be included. Run these two files a few more times (the ingest application followed by the read one) and see how you're not simply getting additional rows (the `cbt_ingest.py` application doesn't add a row; it directly creates one with a specified row key), but you get additional timestamped *versions* of the row.

    Feel free to explore these Python files to see what they're doing. In short, we wrote a row with a column family of three columns, and then we've read it by querying the table using the row key.

13. When you're done exploring, clean up all resources for this chapter by deleting the project (and everything in it):

```
$ gcloud projects delete PROJECT_ID
```

You've now gotten some hands-on experience with Bigtable, a service you will be likely to work with if you're working with data-driven organizations in Google Cloud. Because so many aspects of such a complex service are taken care of, you should regard it as a great option in your toolbox of data solutions as an architect. Remember the well-fit use cases mentioned at the beginning of this section and analyze your workload requirements against other options.

Let's now briefly look into a few of the other non-relational services and what they're best used for.

Cloud Firestore and Firebase Realtime Database

Cloud Firestore is a database option to consider if you need to store highly structured objects (not necessarily with an enforced schema) in a document database, supporting ACID transactions and SQL-like queries. This is for cases when your data is not necessarily relational, but it may require transactions over at least a subset of the data. In addition, it is especially well suited for **mobile applications**, with features such as **real-time updates** that allow you to synchronize and update data on any connected device. Cloud Firestore is the next generation of a service formerly known as *Cloud Datastore*.

Cloud Firestore is a *document-oriented* database where you store data in documents that contain fields that map to values (and may have nested structures). Documents are stored in collections, which are used to organize documents and over which you can build queries. For example, you can have a collection called *books*. Then, inside that collection, you can have several *documents*, each representing a book. These would have fields such as *Title* and *Category*. Since there are no schemas to follow, fields and data types can differ between entities. You can also have sub-collections. Using the same example, we can organize the *book* collection by sub-collections representing publishers: a sub-collection named *Packt* under the *books* collection holds books published by Packt, whereas other publishers would have their own books grouped under their own sub-collections.

This service has mobile SDKs (for Android and iOS) in addition to web SDKs in several languages and frameworks. Cloud Firestore is a serverless service, in that you don't have to deploy a database server or any intermediary server between clients and the database. Clients can connect directly through the SDKs and APIs. Security rules can be defined declaratively (in code) to control access to the database.

Data can be stored in a multi-regional setting (spread across different regions) or in a regional setting (stored within a region). A multi-regional deployment will deliver a higher availability SLA of 99.999%, whereas a regional deployment delivers a 99.99% availability SLA. Data is automatically replicated across multiple regions (in multi-regional deployments) or multiple zones (in regional deployments). In Cloud Firestore, backups are handled by the managed export and import service, with which you can programmatically export all data into a cloud storage account, and later retrieve it through an import operation.

Firebase's Realtime Database is a similar service to Firestore. Both come from Firebase, Google's platform for mobile and web application development, but Firestore is newer and has a richer feature set than Realtime Database. Firestore also scales better and has faster queries. However, for simpler solutions and for specific ways of working with this type of database, you might find Realtime Database to be a better solution. Comprehensive details regarding the differences and a great comparison survey to help you decide which option to use is available at `https://firebase.google.com/docs/firestore/rtdb-vs-firestore`.

Cloud Memorystore

This is Google Cloud's offering for an in-memory data store service for **Redis** and **Memcached**. Tasks including provisioning, replication, failover, and monitoring are all automated and packaged into this PaaS offering, in contrast to a self-hosted Redis solution, for example.

Cloud Memorystore is a fast, in-memory store, and therefore well suited for applications that require fast, real-time processing of data. Common examples of what could benefit from this storage service include user session management, frequently accessed queries and pages, and scripts. Gaming and other such real-time applications can leverage Cloud Memorystore to store, for example, an in-game leaderboard or player profiles. A stream of data such as Twitter feeds or telemetry data from IoT devices can also fit nicely into a Memorystore solution to deliver fast and low latency data access.

Memorystore has two service tiers: **Basic** and **Standard**. The Basic tier can be a good fit for applications that can sustain some loss of data through a cold restart and full data flush. This would most often be the case for applications that use caching, mostly for reading data already stored somewhere else (and reading it faster), or data that is not crucially important to keep.

The Standard tier provides high availability using replication and automatic failover and is well indicated when data durability is of importance. You should remember, however, that this type of in-memory data storage still comes with a trade-off of speed over durability, so even the Standard tier should not be considered if your data requirements set durability as the highest priority.

In terms of technical differences, there aren't many. Both tiers provide instances with up to 300 GB storage size, and a maximum network bandwidth of 12 Gbps. In the Standard tier, each Memorystore instance is configured automatically as a primary and replica pair in an active-standby setup. The replica is always in a different zone than the primary, and data changes made on the primary instance are copied to the standby instance using the *Redis asynchronous replication protocol*. If the primary node fails, the platform will trigger a failover automatically, at which point the replica is promoted to be the new primary and, after recovery of the failure, the former primary becomes the new replica.

When it comes to backups, Memorystore data can be exported to cloud storage and restored at a later point. It is possible to automate this process by using Cloud Scheduler (a service we will discuss in *Part 3*, *Designing for the Modern Enterprise,* of this book) and Cloud Functions. A tutorial on how to do that is available at `https://cloud.google.com/solutions/scheduling-memorystore-for-redis-database-exports-using-cloud-scheduler`.

Cloud storage for unstructured data

Cloud storage is a service we have already mentioned a few times in this book. It's a highly durable object storage service where you can store just about any type of data, including binaries such as image and video files or scripts and executable files. Indeed, you can host a static website on cloud storage by simply placing HTML and accompanying CSS and/or JavaScript files in there and making it publicly accessible.

This service provides different access tiers, or *storage classes*, based on how often data is accessed. Frequently accessed data should use the standard storage class, whereas nearline, coldline, and archive storage classes offer progressively lower storage prices with correspondingly higher retrieval costs (as discussed in more detail in *Chapter 2*, *Mastering the Basics of Google Cloud*, in the section relating to cost effectiveness).

Data in cloud storage is contained in **buckets**. Buckets organize data and centralize access control and some settings for data **objects** in them. Buckets cannot be nested and, because there are limits to bucket creation and deletion, your applications should be designed to *mostly work with object-level operations and not many bucket operations*. A bucket is created in a geographic location you specify, and with a default storage class for objects stored in the bucket (for when objects don't have a specified storage class themselves). Objects are the individual *blobs* or units of data that you store in cloud storage. The service is highly scalable and there is no limit on the number of objects that you can create in a bucket. You pay for the amount of storage consumed.

The geographical location of a bucket can be configured to be a single GCP region, a pair of regions (dual-region setup), or several regions within a broad geographical area (multi-region setup). As of the time of writing, Asia, Europe, and US are the multi-region locations available. Irrespective of the storage class, objects stored in dual- or multi-region buckets will have *geo-redundancy*. Geo-redundancy occurs asynchronously, but all cloud storage data is replicated within at least one geographic location as soon as you upload it.

An attractive feature of cloud storage is *object versioning*. When enabled, an object will have different versions as you modify it (a *modification* being a new object added with the same name and in the same place. In fact, objects in cloud storage are immutable and cannot change throughout its lifetime). This allows you to restore an object that was accidentally replaced, for example, or restore it to a previous *state*. Older versions can be permanently deleted if you no longer need them (the presence of multiple versions will increase the storage costs). The **Object Lifecycle Management** feature allows you to set a **time to live** (**TTL**) for objects and also things such as the number of non-current versions to maintain. You can set a policy to downgrade storage classes and save costs on infrequently accessed data. This feature is very useful for automating clean-up and housekeeping processes that very often get forgotten and, as a result, incur unnecessary storage costs. Life cycle management configurations can be assigned to a bucket, thus applying to all current and future objects in the bucket. Here are examples of rules that you can define as life cycle management policies:

- Delete objects created before January 1, 2018.
- Keep only the two most recent versions of each object that has versioning enabled.
- Downgrade the storage class of objects older than 90 days to nearline storage.

Cloud storage data (in other words, objects) are generally unstructured, although you could add any type of structured or semi-structured data as well, for example, tables or JSON documents. In any case, an object's data is *opaque* to cloud storage, in other words, it's just a chunk of data into which the platform has no visibility. This means you can't run queries or analyze data that is in cloud storage, therefore making it a *storage* service to be used for this exact purpose: storage. It's not a non-relational database, it's an opaque, unstructured object store. To secure your data, server-side encryption is used to encrypt the data by default, but you can also use your own encryption keys. Authentication and access controls can be applied to further secure access. Data security will be a topic revisited in *Chapter 7*, *Designing for Security and Compliance*.

Cloud storage has a command-line utility called `gsutil`, with which you can interact with the service's API. With `gsutil`, you can perform a range of bucket and object management tasks, such as create or delete buckets, upload or download objects, copy and move objects, and edit a bucket's ACLs. It also supports running *resumable uploads* for more reliable data transfers from, for example, on-premises data storage. Some example commands that you can run include the following:

```
# Create a bucket
gsutil mb gs://BUCKET_NAME
# Upload an object
gsutil cp OBJECT_LOCATION gs://DESTINATION_BUCKET_NAME
# Upload an entire directory tree
gsutil cp -r dir gs://DESTINATION_BUCKET_NAME
# Enable object versioning on a bucket
gsutil versioning set on gs://BUCKET_NAME
# Make all objects in Bucket publicly readable
gsutil iam ch allUsers:objectViewer gs://BUCKET_NAME
```

Can you guess how you could use the commands listed previously to host a static website on GCP? If you're up for a challenge, try to host a static website on cloud storage by uploading files to a bucket and making them publicly readable. The `gsutil` command-line utility should be available on your local computer if you've installed the Cloud SDK, but it is also available on Cloud Shell. You can get static website templates for free from `https://html5up.net/`. Once you download one, create a bucket and upload the local directory tree to it, and then set an IAM policy at the bucket level so that all users can view its objects. You won't need to run any additional commands other than the ones listed previously. Once you've finished, you can navigate via your browser to `https://storage.googleapis.com/BUCKET_NAME/DIR_NAME/index.html` (replacing `BUCKET_NAME` and `DIR_NAME` with the name of your cloud storage bucket and the directory you've uploaded, respectively). To delete a bucket, you can run the following command:

```
gsutil rm -r gs://BUCKET_NAME
```

If you're familiar with Linux and Bash commands, you may have noticed that `gsutil` commands are built with very similar syntax. That makes it easy to guess how to run a command based on how you would perform the same object manipulation on a local, Linux-based filesystem.

Consider cloud storage for your unstructured object storage needs, as well as a place for database backups, or data that is to be archived or accessed infrequently (in which case, you can leverage cost savings with storage classes). If you're hosting websites on GCP, you can place some of the static contents in cloud storage, such as JavaScript and CSS files. If the website is entirely static, you can place all of it in there and make it publicly accessible, effectively delivering a highly available and scalable website in a cost-effective manner. Choose at least a dual-region setting for higher availability and durability, but leverage a multi-region deployment when your service and your users are globally distributed or if you need very high durability. Finally, consider enabling object versioning as insurance against accidental object replacement, and don't forget the Object Lifecycle Management feature as a way to save costs and automate housekeeping of your objects.

Choosing the right solution for each piece of data

As mentioned previously, a cloud design that incorporates polyglot persistence, where different data services and technologies are used to meet each specific data type and its requirements, is ideal for modern applications with even modest levels of data complexity. Therefore, you will likely find yourself having to make several decisions when designing the data architecture for your solutions. The following diagram summarizes what you've learned in this chapter into a decision tree to guide you in choosing the best storage solution according to the characteristics of the data:

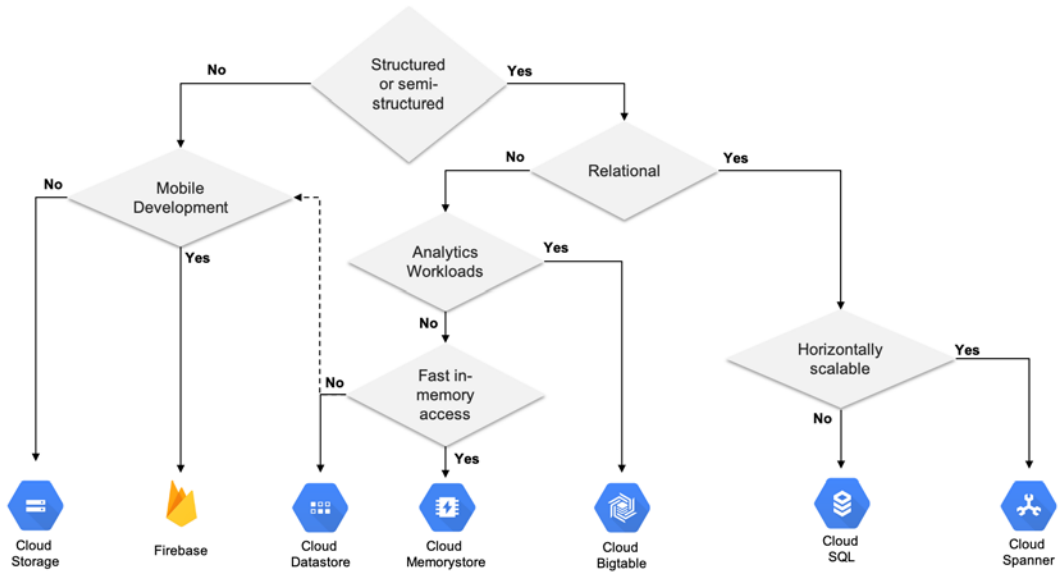

Figure 5.13 – Decision tree for GCP data services

By asking yourself just a few questions about the qualities and the needs of your data, you should be able to find a suitable datastore on GCP, at least as a starting point. There is one data service that was left out of this chapter, and that was Cloud BigQuery. This service, as well as big data concepts such as data lake, data warehouse, and data pipelines, will be discussed in *Chapter 8*, *Approaching Big Data and Data Pipelines*.

Summary

In this chapter, we covered several foundational concepts regarding data and storage solutions. We then explored relational and structured data services in Google Cloud, and how to design those services for high availability and scalability, and how to handle backups and approach data durability and consistency. You got some hands-on experience with Cloud Spanner, a horizontally scalable relational database service that is also highly available. Finally, you learned about different services for non-relational or unstructured data in Google Cloud. In most modern solutions, a cloud architect will need to be prepared for *polyglot persistence*, a mix of different types of databases for different types of data. Therefore, having learned about all those services will help you design robust cloud-based data architectures that leverage polyglot persistence to obtain optimal solutions.

In the next chapter, we're going to look into one of the strong pillars of a well-architected cloud solution: *observability*.