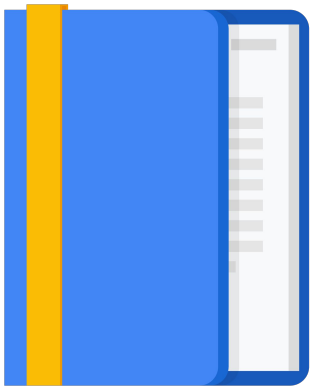
Agenda

Solution Overview

Traditional Database Architectures

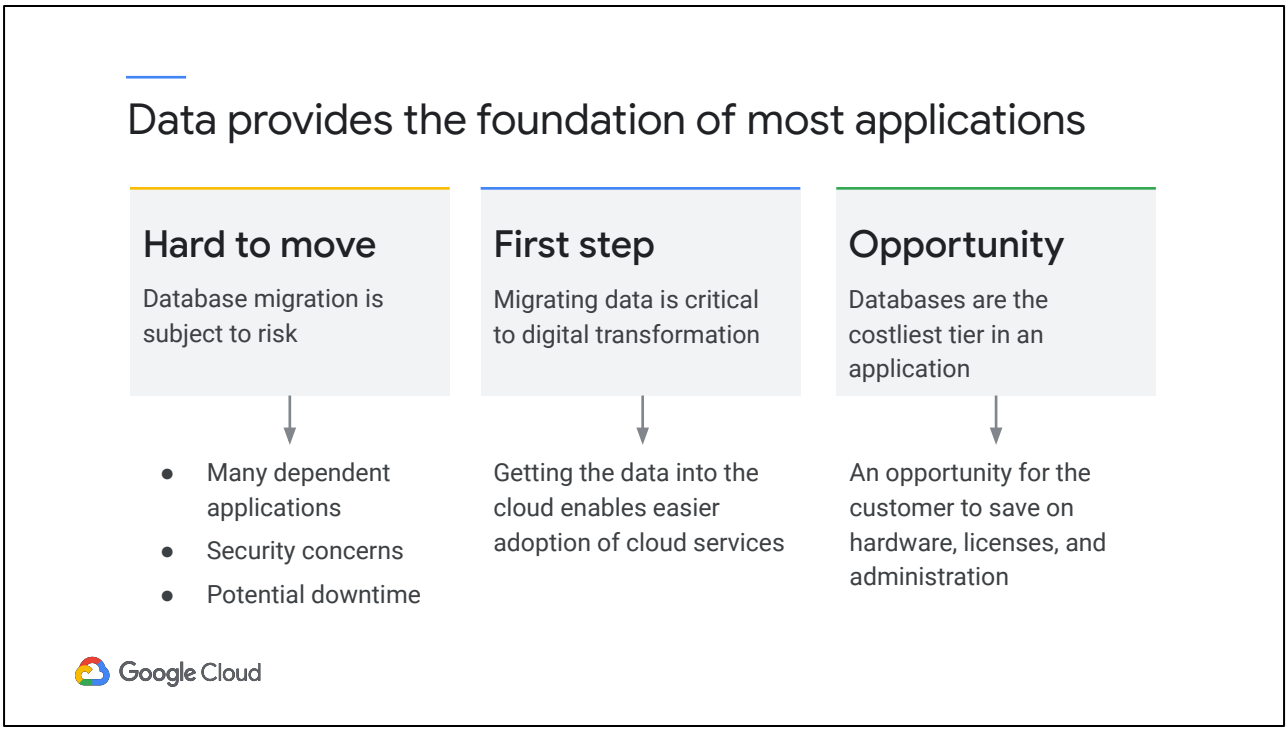
Optimizing Databases for the Cloud

Architecting Scalable and Highly Available Databases



Google Cloud

Let's get started with an overview of database migration problems and some solutions provided, not only by Google Cloud, but other providers as well.



- Databases can be the hardest part of an application to move. Often, many dependent applications rely on the database for their data. And those applications may be constantly adding new data to the database. Moving the database can break those connections.

  There are also security concerns. When hackers are attacking your systems, they are usually after your data. You need to design and architect your system in a way that protects the database, but allows dependent applications to have access.

  For some applications, defining a maintenance window and bringing the database down for that window of time is acceptable. For mission-critical applications though, downtime needs to be minimized and sometimes avoided completely. This complicates your migration project significantly. Automation, database replication, and extensive testing will be required to ensure that when you “flip the switch” to move from the old database to the new one, everything will continue to work seamlessly.

- Because the database is so important to the operation of its dependent applications, moving it to the cloud is often a critical first step when migrating applications. After the database is moved and running, moving other applications is comparatively easy.

  Customers often want to take advantage of the many services Google

provides as part of a digital transformation. When you’re running in the cloud, you can more easily take advantage of Google's advanced machine learning and big data processing services, for example.

- Moving the database can also be a big cost-saving opportunity for customers. Between hardware, maintenance, licensing, and administration, the database tier is likely the most expensive part of an application. Moving to Google Cloud can help reduce all of those costs. When their database is in the cloud, customers can work to optimize their databases and move to even cheaper, completely managed data services like BigQuery, Firestore, and Spanner.

Database is Hard to Move

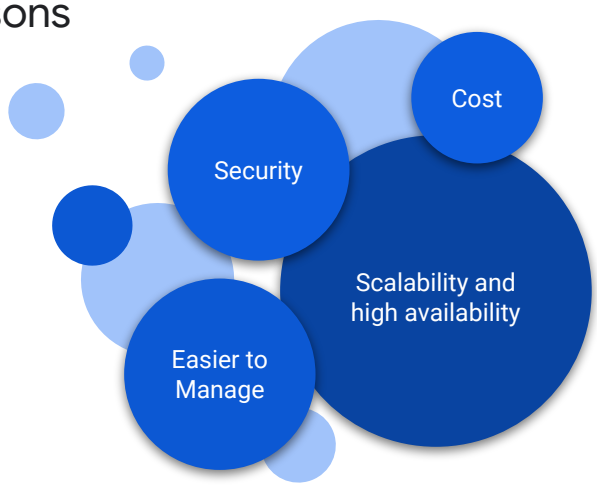
- We require
  1. Automation
  2. Database Replication
  3. Extensive Testing
- to ensure: when “flip the switch” to move from old to new DB,
  - everything will continue to work seamlessly

Moving Database can be a Big Cost-saving

- Save on
  1. hardware
  2. maintenance
  3. licensing
  4. administration

- When their database is in the cloud, customers can work to
  - optimize their databases
  - move to cheaper, completely managed services like BigQuery, Firestore, Spanner

Customers choose to move databases to the cloud for many reasons



Google Cloud

There are many great reasons customers want to move their databases and applications to Google Cloud.

- Google has data centers in regions all over the world, and each region is divided into multiple zones. You can deploy your database to multiple zones or even multiple regions for greater scalability and fault tolerance.
- Google will manage all the hardware for you, which simplifies the management of your databases. If you use a managed database service like Cloud SQL or Spanner, Google does practically all the maintenance for you.
- Sometimes people think moving to the cloud is less secure. On the contrary, Google's security is unmatched. If you know what you are doing, moving to Google Cloud can in fact enhance the security of your applications and data.
- And of course, decreasing the total cost of ownership of running your applications is often a primary driver for moving to Google Cloud.

Scalability vs Highly Available (HA)

- **`Scalable System`**
  - continues to work even as the number of users and the amount of data grow

- **`Highly Available System`**
  - continue to work even when there is a failure

Durability

- **In materials**
  - the ability to remain serviceable in the surrounding environment during the useful life without damage or unexpected maintenance

- **In databases**
  - once a transaction completes successfully,
    - its effects cannot be altered without running a compensating transaction
  - changes made by successful transaction survive subsequent failures of the system

Fault Tolerance vs High Availability (1/2)

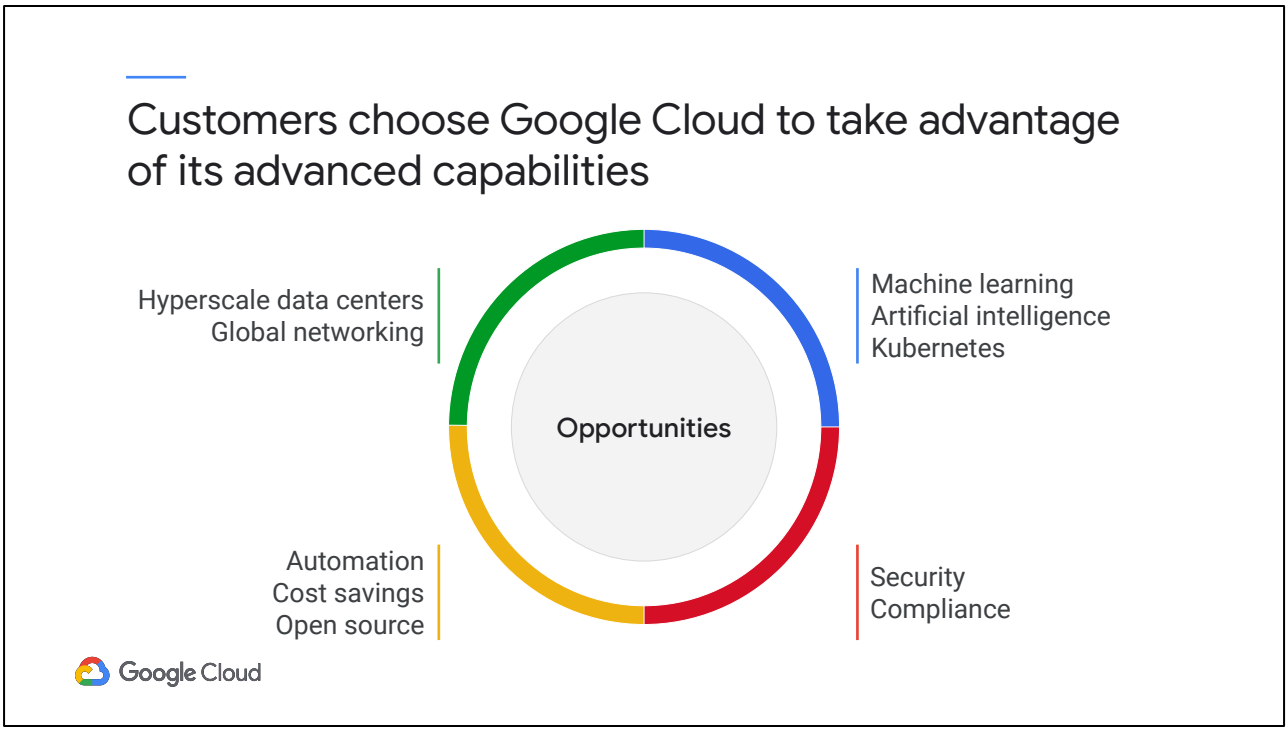
- **Fault Tolerant Environment**
  - relies on **specialized hardware** to detect **hardware fault**
  - instantaneously switch to a **redundant hardware** component
  - has no service interruption but a significantly higher cost
  - does not address **software failures**

- **Highly Available Environment**
  - services are restored rapidly (often in less than a minute) but not instantaneous
  - has a minimal service interruption
  - are excellent solution for apps that must be restored quickly
  - combines **software** with **hardware** to minimize downtime

Fault Tolerance vs High Availability (2/2)

- Many sites
  - can absorb a small amount of downtime with **HA**
  - rather than pay the much higher cost of providing **Fault Tolerance**

- Some industries
  - have apps so time-critical and cannot withstand even a few seconds of downtime
- Many other industries
  - can withstand small periods of time when their **database** is unavailable



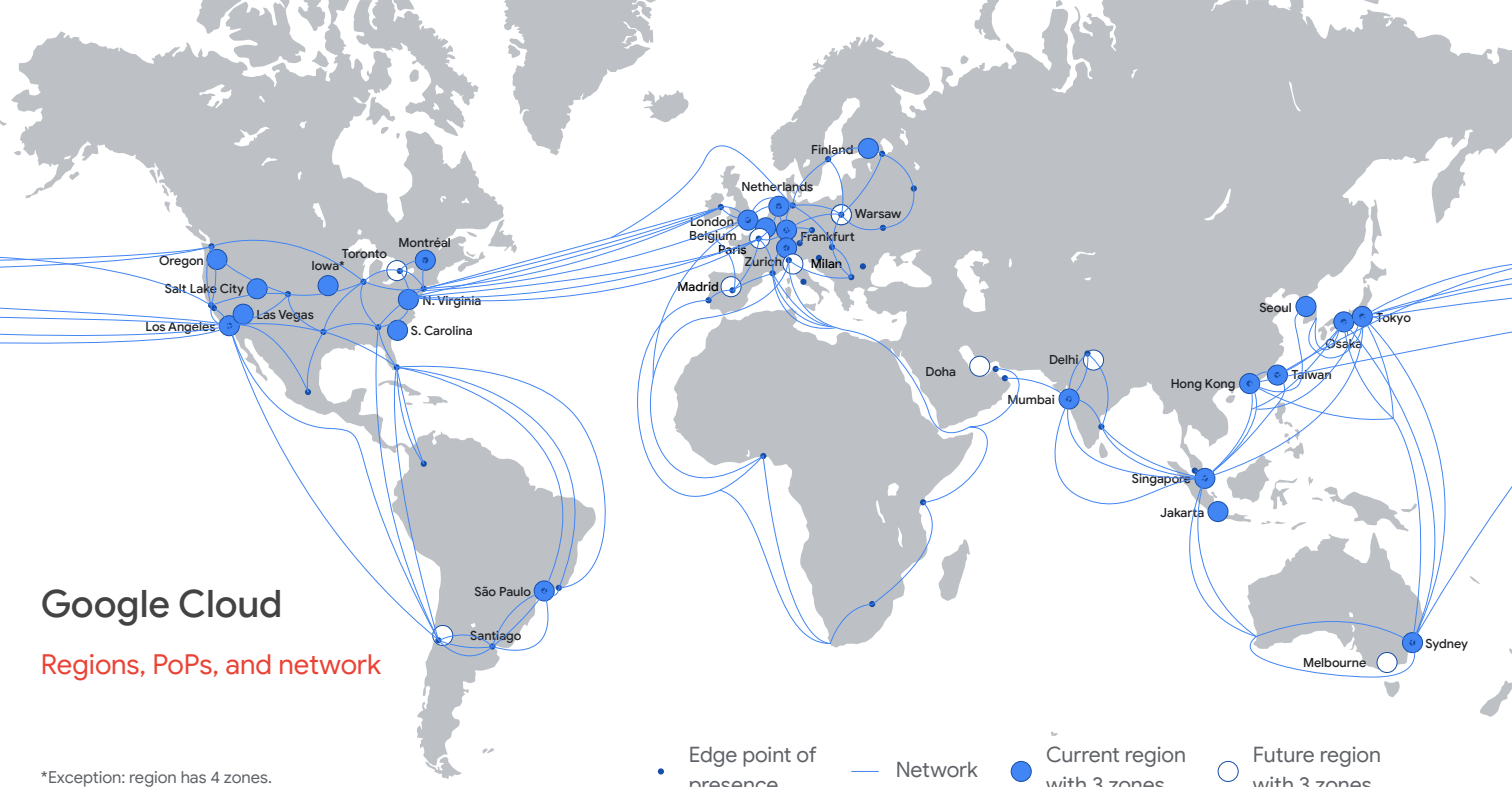
Google Cloud offers advanced capabilities that only the largest organizations could replicate in their own data centers.

- There is practically unlimited compute power and storage provided by Google's many hyperscale data centers located all around the globe. The data centers are connected by Google's fast and reliable global network.
- All resources in Google Cloud can be automated for easier management and cost savings. Google also embraces open-source technologies and is a big contributor to the open-source community. Google uses a custom Linux distribution for its own servers. Some very popular open-source technologies like TensorFlow and Kubernetes originated at Google.
- Google is a leader in advanced technologies, which is often why customers choose Google over other cloud providers. Some customers want to add machine learning capabilities to their applications using TensorFlow or use one of Google's artificial intelligence APIs. Many customers want to simplify their data center management using Kubernetes.
- Enhanced security is also an important factor when customers move to the cloud. Because Google Cloud is already certified by many government and industry compliance standards, running on Google Cloud can make compliance easier for many customers.

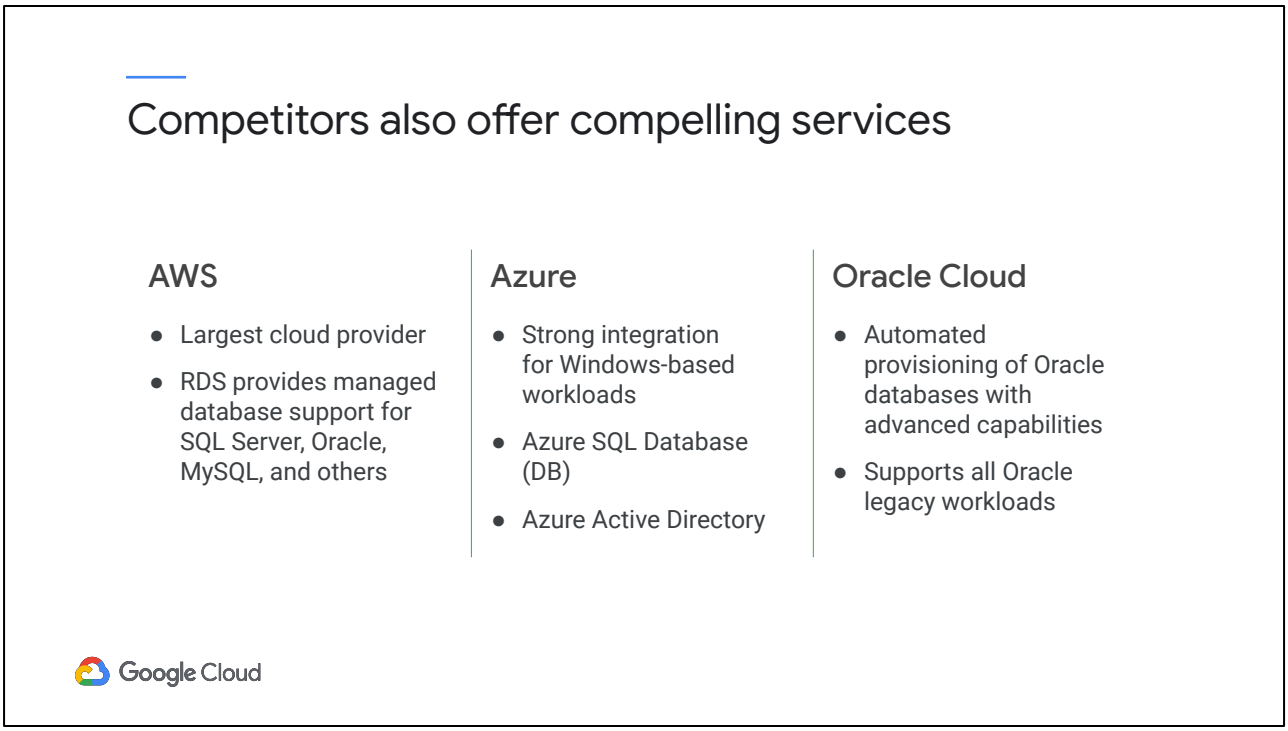
Google Cloud

- Resources (including databases) can be automated for
  - easier management
  - cost savings

- Google
  - embraces open-source technologies
  - is a big contributor to the open-source community
  - uses a custom Linux distribution for its own servers
  - created some very popular open-source technologies like TensorFlow and K8s



*Exception: region has 4 zones.



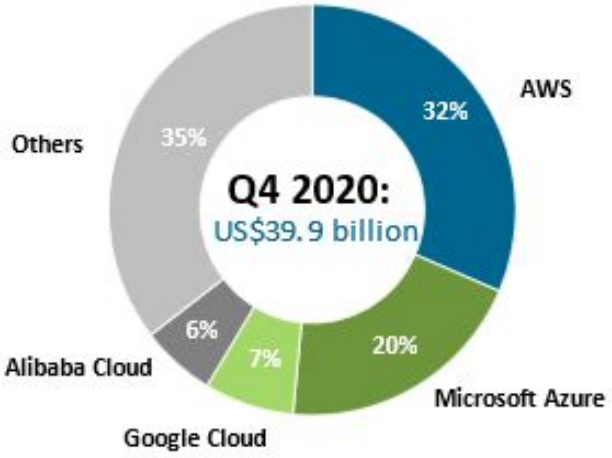
Other cloud providers, competing with Google, also offer compelling platforms and services.

- **Amazon Web Services** is one of the largest cloud providers in the world. They offer a managed database solution called RDS. RDS supports SQL Server, Oracle, MySQL, and other databases. Amazon also has a customized version of MySQL called Aurora that is optimized to run on their cloud. RDS has many advanced features like automated backups and automatic replication for high availability.

- For customers who rely heavily on Windows, **Microsoft Azure**, as you would expect, provides very strong integration with Microsoft products and tools. Azure SQL Database provides SQL Server as a service. There is also a cloud-based Azure Active Directory service. Azure also provides support for open-source technologies. You can run Linux or Windows virtual machines and many other databases like MySQL and MongoDB. Azure also supports Kubernetes through Azure Kubernetes Service.

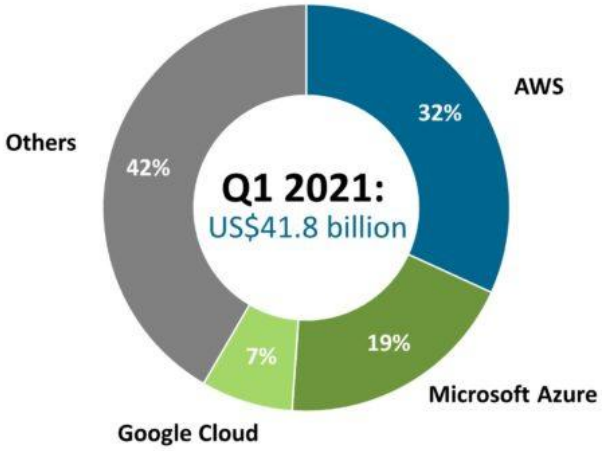
- **Oracle** also provides their own cloud for those customers who rely on Oracle databases. Oracle Cloud automates the provisioning of Oracle databases. It differs from the managed service provided by AWS RDS in that it supports all Oracle features, which RDS does not, and it supports all Oracle versions.

Cloud Providers Market-share



Source: Canalys estimates, February 2021

canalys



Source: Canalys estimates, April 2021

canalys

Amazon Web Services (AWS) 1/2

- **`Amazon Relational Database Service (Amazon RDS)`**
  - web service to facilitate: set up & operate & scale a relational database in the cloud
  - provides cost-efficient, resizeable capacity for industry-standard relational DB
  - manages common database administration tasks (managed service)
  - supports **Oracle** & **SQL Server** & **MySQL** & **MariaDB** & **PostgreSQL**
  - `has many advanced features like`
    - `automated` **`backups`**
    - `automatic replication for` **`HA`**

Amazon Web Services (AWS) 2/2

- **`Amazon Aurora`**
  - part of Amazon RDS
  - fully-managed relational database engine that's built for the cloud
  - **customized versions** of **MySQL** or **PostgreSQL** that is optimized to run on AWS

Microsoft Azure

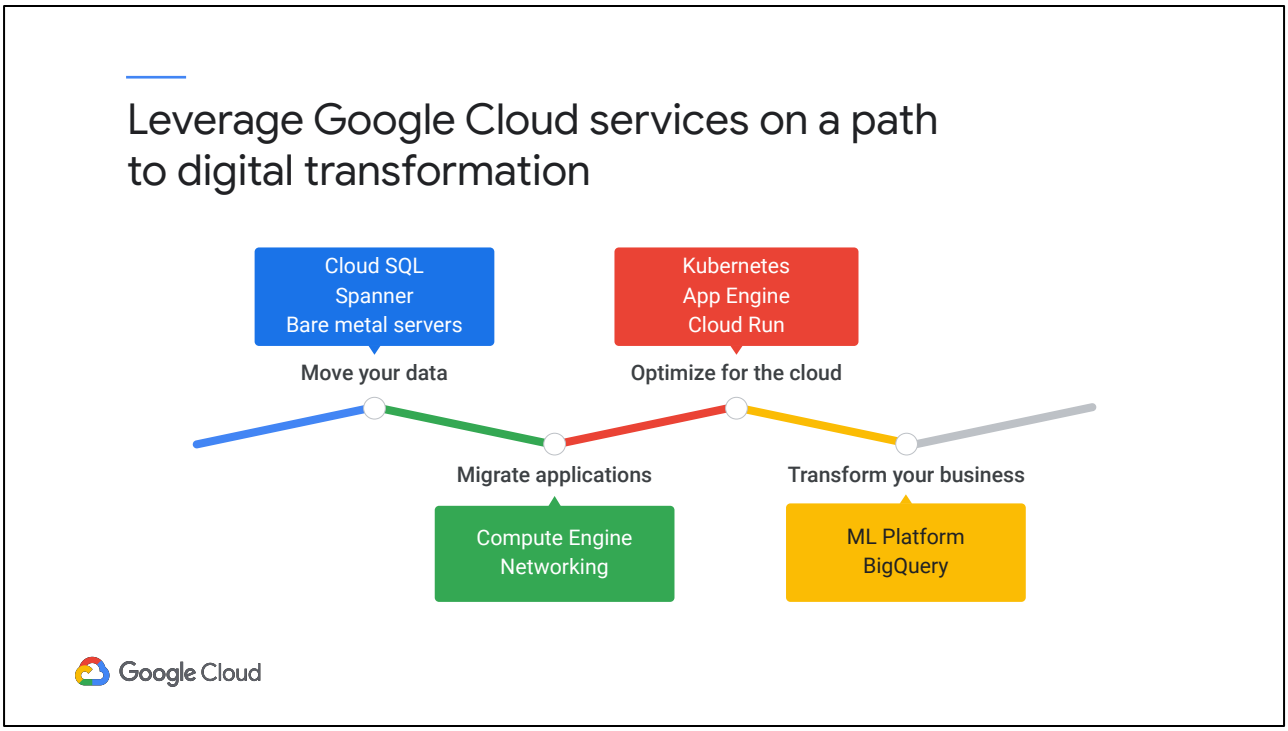
- **Azure SQL Database**
  - SQL Server as a service

- **Azure Active Directory (Azure AD)**
  - cloud-based Active Directory, which:
    - stores info about objects on the network
    - makes this info easy for admins and users to find and use

- Provides support for **open-source** technologies
  - can run many databases like **MySQL** and **MongoDB**
  - can run **Linux** or **Windows** VMs
  - supports **Kubernetes** through **Azure Kubernetes Service (AKS)**

Oracle Cloud

- `differs from AWS RDS in`
  - supports all Oracle features
  - supports all Oracle versions



In summary, customers on a path to digital transformation want to leverage Google Cloud's hyperscale data centers, global reach, and advanced services. That path starts with moving your data and databases, and then moving your applications. After you're in the cloud, you can start optimizing for the cloud and ultimately transform your business with machine learning and artificial intelligence.

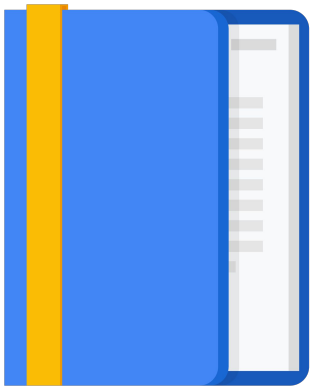
Agenda

Solution Overview

Traditional Database Architectures

Optimizing Databases for the Cloud

Architecting Scalable and Highly Available Databases

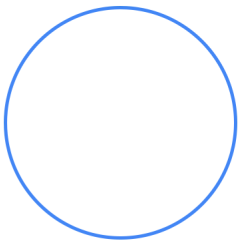


Google Cloud

In this section, you learn about the different database architectures you will encounter. In one sense, this is a history lesson on how database architectures have evolved. However, in large organizations, you might find a mix of database architectures, depending on who created the databases and what they were designed for.

Client-server databases have been a standard since the 1980s

- Highly normalized.
- Business logic provided by the database:
  - Constraints and relationships
  - Stored procedures
  - Triggers
- Client applications connect directly to the database.



Google Cloud

Client-Server databases have been a standard for many years. Client-server databases tend to be highly normalized. Also, business logic tends to be provided by the database server itself, rather than by the client. Business rules are implemented using constraints on fields and relationships between tables, as well as stored procedures and triggers. Clients connect directly to the database.

Clients in a client-server architecture were intended to be as thin as possible. The work of managing transactions and enforcing rules was the domain of the database server where this logic was centrally located and shared by the clients.

Client-server databases are fast and secure and are the preferred architecture for many DBAs.

Client-server Databases

- **Business rules** are implemented using
  - **relationships** between tables
  - **constraints** on fields
  - stored **procedures**
  - **triggers**

- Database server
  - manage transactions
  - enforce **business rules**
  - where **business logic** is located to be shared by the clients

Procedure Example 1 (1/2)

```
mysql> create database courses;
Query OK, 1 row affected (0.01 sec)

mysql> use courses;
Database changed
mysql> create table students (
    -> id int not null auto_increment primary key,
    -> fname varchar(200),
    -> lname varchar(200)
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> insert into students (fname, lname) values ('Ahmed', 'Hany');
Query OK, 1 row affected (0.00 sec)

mysql> insert into students (fname, lname) values ('Mary', 'Ibrahim');
Query OK, 1 row affected (0.00 sec)
```

Procedure Example 1 (2/2)

```
mysql> delimiter //
mysql> create procedure get_all_students()
    -> begin
    -> select * from students;
    -> end
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> call get_all_students() //
+----+-------+---------+
| id | fname | lname   |
+----+-------+---------+
|  1 | Ahmed | Hany    |
|  2 | Mary  | Ibrahim |
+----+-------+---------+
2 rows in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

Procedure Example 2

```
mysql> create procedure get_name (in stu_id int, out name varchar(401))
    -> begin
    -> select fname into name from students where id = stu_id;
    -> end //
Query OK, 0 rows affected (0.01 sec)

mysql> call get_name(1, @name) //
Query OK, 1 row affected (0.00 sec)

mysql> select @name //
+-------+
| @name |
+-------+
| Ahmed |
+-------+
1 row in set (0.00 sec)
```

Procedure Example 3

```
mysql> drop procedure if exists get_name //
Query OK, 0 rows affected (0.01 sec)

mysql> create procedure get_name(in stu_id int, out name varchar(401))
    -> begin
    -> select concat(fname, ' ', lname) into name from students where id = stu_id;
    -> end //
Query OK, 0 rows affected (0.01 sec)

mysql> call get_name(1, @name) //
Query OK, 1 row affected (0.00 sec)

mysql> select @name //
+------------+
| @name      |
+------------+
| Ahmed Hany |
+------------+
1 row in set (0.00 sec)
```

Trigger Example 1 (1/2)

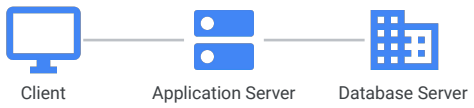
```
mysql> create database bank;
Query OK, 1 row affected (0.01 sec)

mysql> use bank;
Database changed
mysql> create table accounts (
    -> id int not null auto_increment primary key,
    -> name varchar(200) not null,
    -> amount decimal(15,2)
    -> );
Query OK, 0 rows affected (0.03 sec)

mysql> create trigger t1 before insert on accounts
    -> for each row set @sum = @sum + new.amount;
Query OK, 0 rows affected (0.02 sec)
```

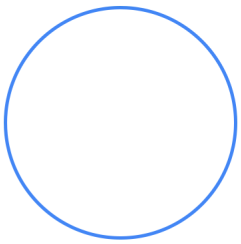
Trigger Example 1 (2/2)

```
mysql> set @sum = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into accounts (name, amount) values ('customer 1', 1000.00);
Query OK, 1 row affected (0.01 sec)

mysql> insert into accounts (name, amount) values ('customer 2', 2000.00);
Query OK, 1 row affected (0.01 sec)

mysql> select @sum;
+---------+
| @sum    |
+---------+
| 3000.00 |
+---------+
1 row in set (0.00 sec)
```

Three-tier architectures pulled the business logic from the database into the application tier

- Developers advocate that business logic is the domain of the code.
- Database become just the storage platform:
  - Fewer stored procedures
  - Performance suffers
- Clients connect to the application server.





Google Cloud

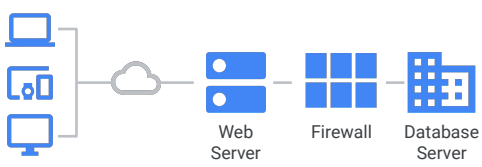
Programmers, especially object-oriented programmers, argued that business logic and transaction management should be the domain of the application, not the database. With a 3-tier or N-tier architecture, much of the business logic is pulled from the database and put into the application code.

The database becomes more of a storage tier; the middle tier handles the complexity. In a 3-tier architecture, there are fewer, if any, stored procedures. While this makes programmers happy, database performance overall can suffer, especially when accessing multiple tables within a transaction.

In this architecture, clients don't connect directly to the database; instead, they connect through an intermediary application server.

Service-oriented architectures expose the application logic over the internet

- Database details are encapsulated by a service.
- Communication is done with HTTP:
  - Data passed between client and server as text (XML or JSON)



Google Cloud

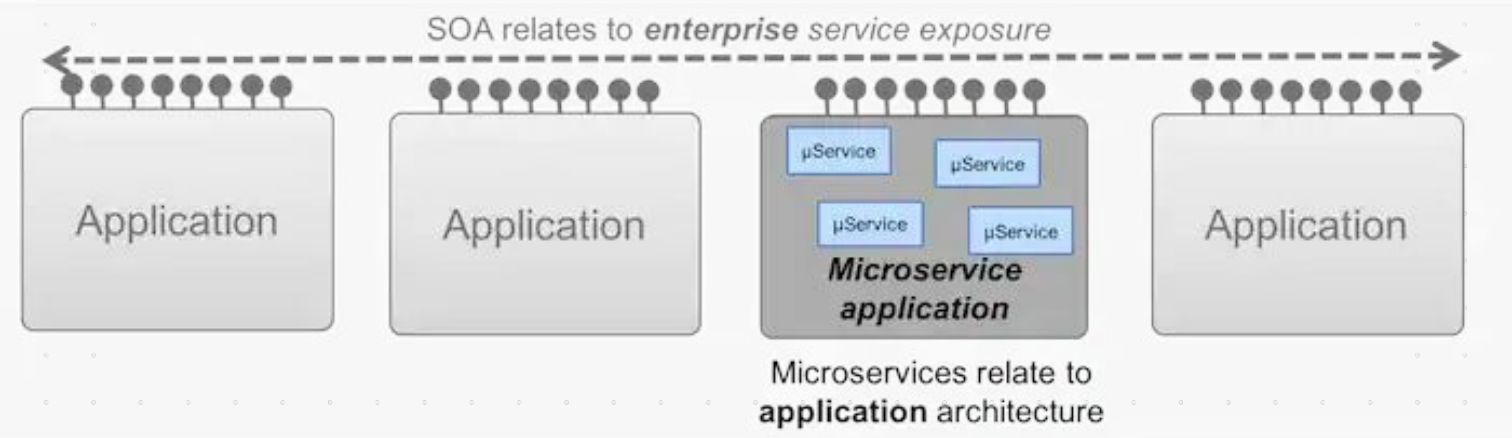
As the internet has grown, and more and more different types of clients have appeared, there has been a trend toward service-oriented architectures. Databases sit behind a firewall, and their functionality is encapsulated, or hidden, behind a service that is made available over the network.

Clients don't need to know anything about the details of the database. Clients only connect to the service via HTTP and pass the data back and forth using a text-based format like XML or JSON.

Only the service makes the connection to the database.

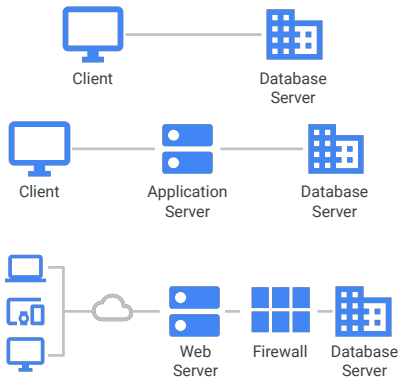
SOA vs Microservices

SOA means Service-Oriented Architecture



In the real world, you'll find a mix of all these architectures

- No one has a pure system.
- Logic is often spread out among all those layers, making it hard to debug and maintain.



Google Cloud

Of course in the real world, you won't find a company that uses one architecture or the other. You'll find a mix of all of these architectures depending on who designed and programmed the systems, and when.

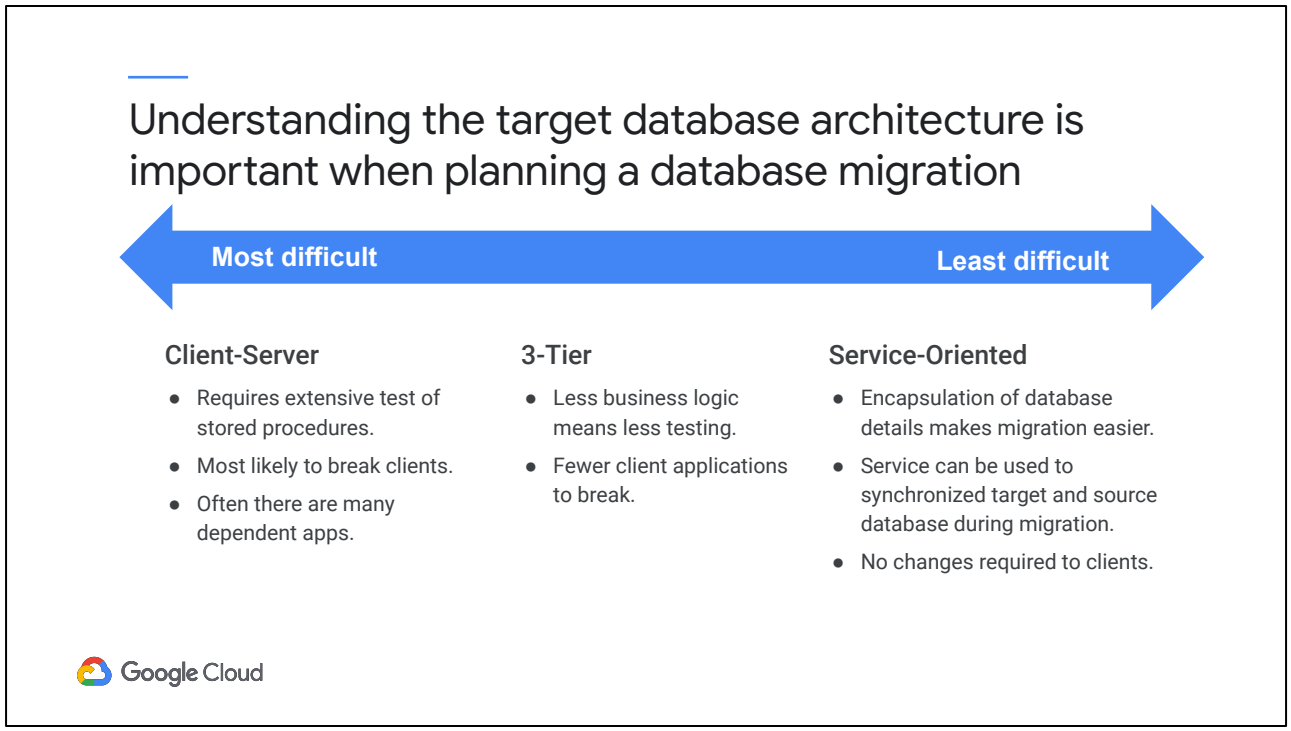
In older applications, you might find a mix of architectures within a single database, depending on when new features and data were added over time.

This can lead to confusion when you are analyzing applications and databases you want to migrate.

A key aspect to migrating databases is finding dependents and dependencies.

A dependent is something that relies on the database or application, and a dependency is something that the application or client must have to function properly.

For example, in a 3-tier architecture, the application server is a dependency for the client. At the same time the application server is a dependent of the database.



Understanding the architecture of the database you are trying to move is important when planning a migration project.

- Generally, **client-server** databases are harder to move because there is more logic provided by the stored procedures that needs to be verified and tested. Any error would probably break the clients, and more dependent applications tend to be connected directly to the database.

- **3-tier** or N-tier applications have most of the business logic in the application code, so there is less testing required in the database tier. Also, there tend to be fewer dependents, because the clients connect to the application server, which in turn connects to the database.

- **Service-oriented** applications can be the easiest to move to the cloud because the database and all its details are hidden behind the service layer. The service can be used to synchronize the source and target databases during the migration. Eventually, the new database takes over and the old one can be turned off. There is no need to worry about the clients because they are simply passing data between themselves and the service.

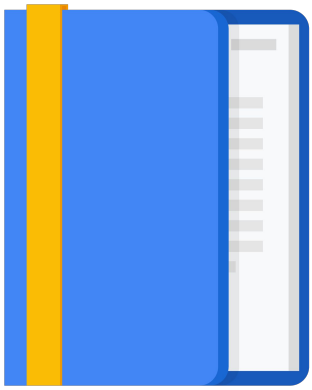
Agenda

Solution Overview

Traditional Database Architectures

Optimizing Databases for the Cloud

Architecting Scalable and Highly Available Databases

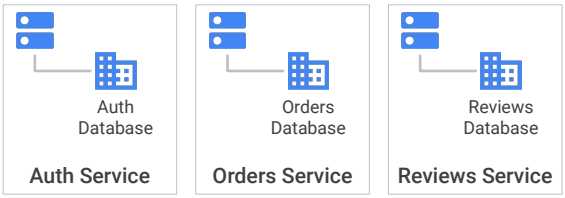


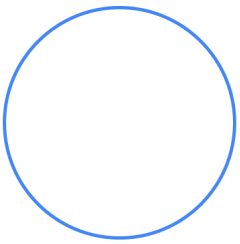
Google Cloud

After you move your databases and applications, there is an opportunity for optimizing them to run in the cloud. Let's briefly talk about how you could do that.

Microservice architectures are the trend when developing cloud-native applications

- Large applications are divided into multiple smaller independent ones.
- Each service has its own database instead of one big master database.





Google Cloud

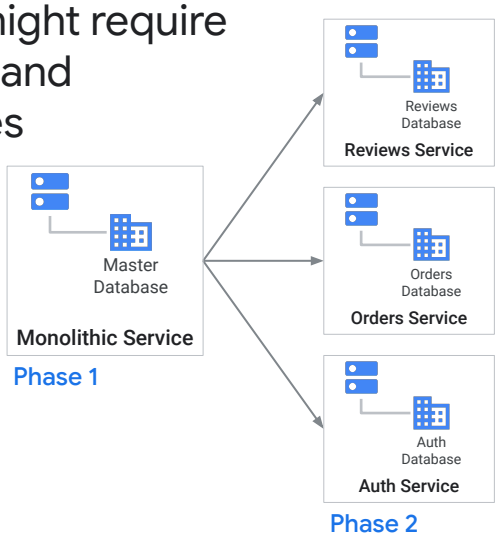
Microservice architectures have become popular as more and more applications have been migrated to the cloud.

With microservices, large applications are divided into smaller, simpler, independent services. Each microservice should be responsible for its own data. This is a big change. You want to design your services to minimize the dependencies between them. Microservices should be loosely coupled. That means one service can use another without understanding its details. Data passed between services is simply text in JSON or XML format.

Also, different microservices should be able to be deployed and versioned independently. Recently, a customer complained that they were having trouble synchronizing the deployment of three services because they all had to be deployed at the same time for the application to work. This is definitely an anti-pattern. In this case, the services are too entangled; they need to be either refactored or combined.

Optimizing for the cloud might require splitting large applications and databases into smaller ones

One large database serving many microservices is an anti-pattern; it complicates the deployment architecture without realizing the benefits of a microservice design.

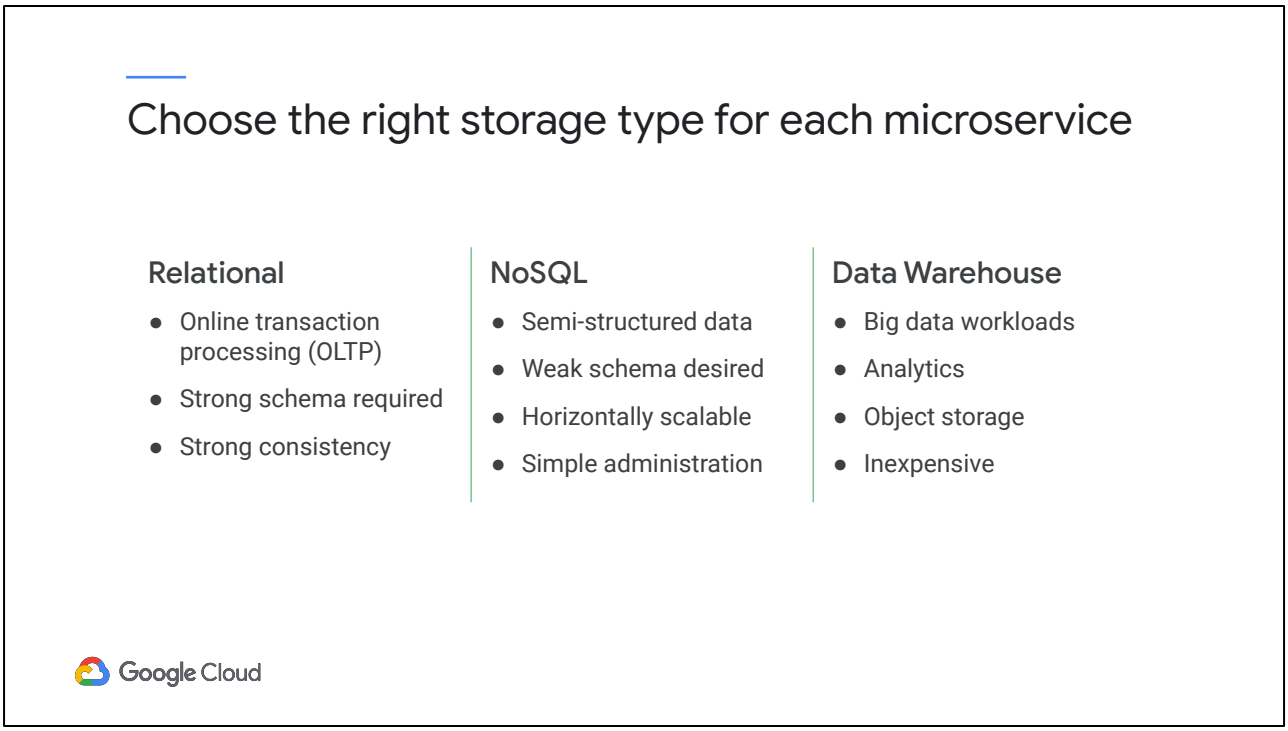


Google Cloud

Because each microservice should be responsible for its own data, when applications are optimized for the cloud, databases also have to be split into multiple, smaller pieces.

Some developers make the mistake of having one main database service provide the data to all the microservices in an application. This complicates your deployment without providing all the benefits of a microservice architecture.

There are many benefits of microservices. One benefit is the ability to choose which type of database is most appropriate for each individual service. For some services, a relational database is required. But for others, using a NoSQL database or a data warehouse can save money and make programming easier.



With monolithic apps, the database you choose has to be one that works for the entire application. That means you probably need a relational database. Relational databases are great, and they work for nearly any application. However, relational databases tend to be more expensive and require more administration than other, simpler types of databases. With microservices, you can keep the relational database when the service requires it, but you can use NoSQL databases or data warehousing solutions when they are a better fit for the microservice use case.

- Use **relational** databases for online transaction processing use cases when strong schemas and strong consistency are required.
- **NoSQL** databases can be simpler and less expensive than relational systems. Use a NoSQL database when you prefer or can tolerate a schemaless database or when a higher degree of flexibility is needed.
- If a microservice is for big data, analytics, or object storage, consider a **data warehousing** solution like BigQuery or Cloud Storage. These solutions are extremely inexpensive and support unlimited amounts of data.

Database Schema

- Defines how data is organized within a relational database, including:
  - table/fields names
  - data types
  - relationships
  - ...

- Considered the "**database blueprint**"
  - the schema does not actually contain data.
- Commonly use visual representations
  - to communicate the architecture of the database

If a Microservice is for

- big data or analytics (using SQL), consider data warehousing solution as **BigQuery**
  - **BigQuery:** `inexpensive and support Petabyte of data`
- object storage, consider **Cloud Storage**
  - **Cloud Storage:** `inexpensive and support Petabyte of data`
  - `object storage vs block storage vs file storage?`
- wide-column NoSQL database, consider **Bigtable** (similar to Cassandra)
  - **Bigtable:** `support Petabyte of data`
- in-memory database, consider **Memorystore**
  - **Memorystore:** support **Redis** and **Memcached** databases

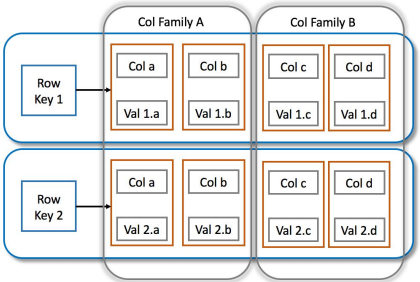
Wide-column NoSQL Database

- Type of NoSQL database in which the names and format of the columns can vary across rows, even within the same table

- Also known as **Column Family Databases**
  - related columns can be modeled as part of the same column family

- Because data is stored in columns
  - queries for a particular value in a column are very fast
  - as the entire column can be loaded and searched quickly

- As **Apache Cassandra** and **ScyllaDB**



Redis vs Memcached

	Memcached	Redis
Sub-millisecond latency	Yes	Yes
Developer ease of use	Yes	Yes
Data partitioning	Yes	Yes
Support for a broad set of programming languages	Yes	Yes
Advanced data structures	-	Yes
Multithreaded architecture	Yes	-
Snapshots	-	Yes
Replication	-	Yes
Transactions	-	Yes
Pub/Sub	-	Yes
Lua scripting	-	Yes
Geospatial support	-	Yes

Database-as-a-service products reduce administrative overhead and cost

Google Cloud

Google Cloud provides a complete collection of database and data storage solutions.

There are two managed database services: Cloud SQL and Spanner. Cloud SQL allows you to run MySQL, PostgreSQL, and SQL Server databases. Spanner is a Google-created relational database service that is massively scalable and can easily be deployed across multiple regions for extremely high availability and low latency around the world.

Google provides a completely managed, highly scalable and inexpensive NoSQL database solution called Firestore. Memorystore provides a managed Redis database solution, and Cloud Bigtable is a managed wide-column NoSQL database similar to Cassandra.

For data warehousing and analytics using SQL, use BigQuery. Use Cloud Storage for cheap object storage.

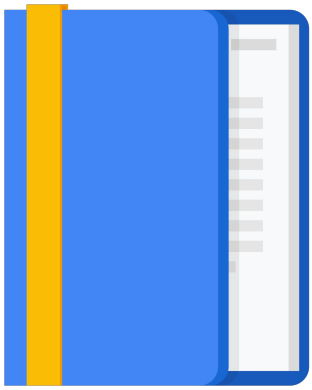
Agenda

Solution Overview

Traditional Database Architectures

Optimizing Databases for the Cloud

Architecting Scalable and Highly Available Databases

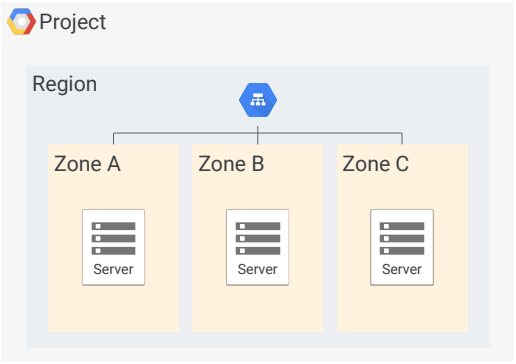


Google Cloud

A system that is scalable is one that continues to work even as the number of users and the amount of data grow. Highly available systems are fault-tolerant and continue to work even when there is a failure of some of the nodes. Let's talk about how you can architect your databases to achieve scalability and high availability.

High availability is achieved by deploying infrastructure across more than one zone in a region

- In the cloud, a zone is considered a fault boundary:
  - Applications deployed to multiple zones continue to work even if a zone fails.
- Route traffic to application instances through a load balancer:
  - Load balancers are regional (*or sometimes global*) resources.
  - Load balancers would survive a zonal outage.



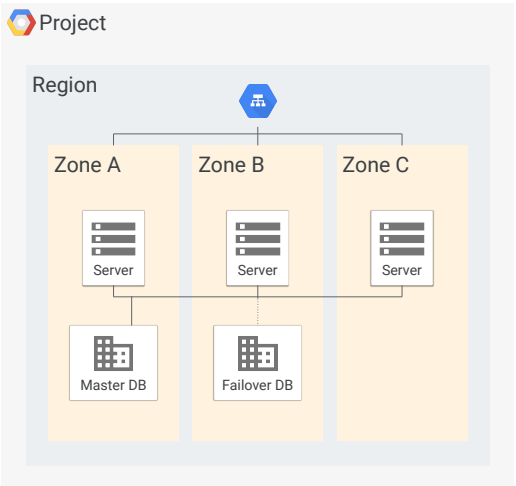
Google Cloud

High availability is achieved by deploying infrastructure across more than one zone in a region. In Google Cloud, each region is divided into three zones. Think of a zone as a fault boundary. No single thing that can fail would cause resources deployed in different zones to be unavailable.

If you're deploying an application like a web app or service, create instances in multiple zones, and then create a load balancer that routes traffic to those backend instances. The load balancers will monitor the health of the instances and only send traffic to healthy ones. The load balancers are services provided by Google and are also fault-tolerant. Load balancers can be either regional or global. In either case, they will survive a zonal outage.

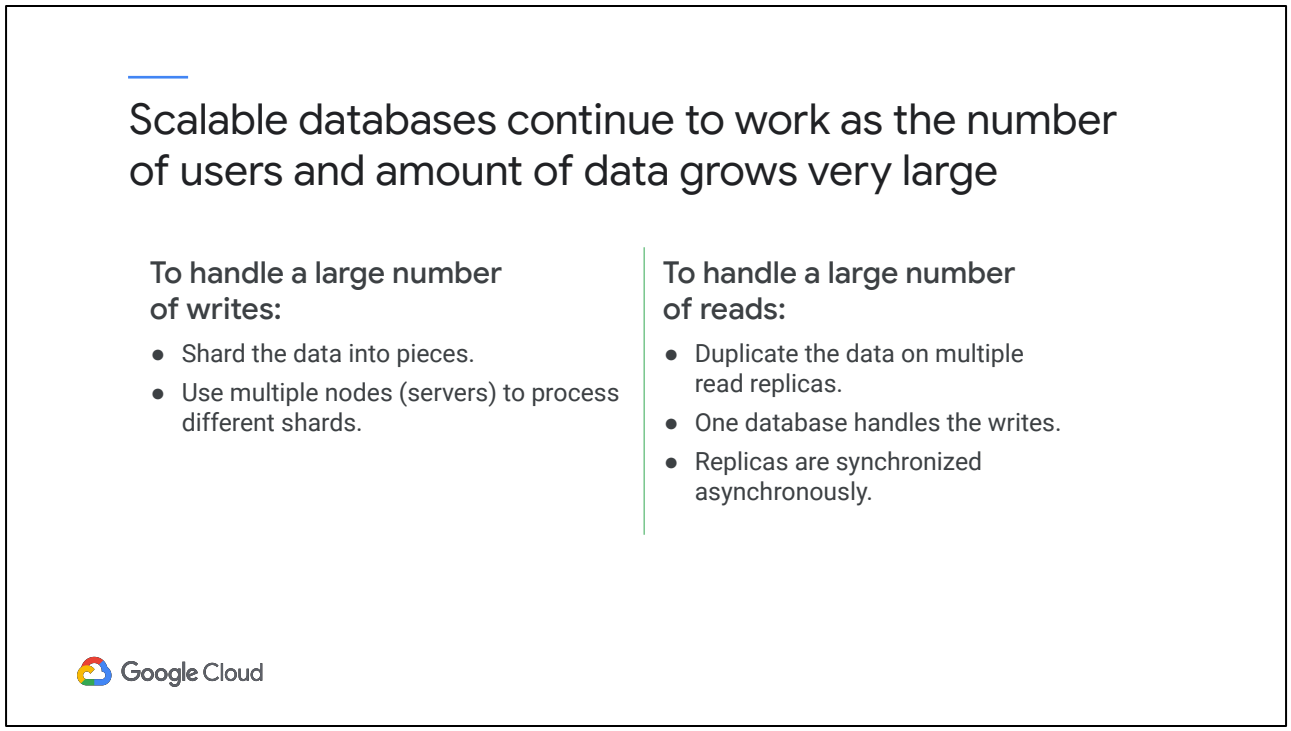
Failover replicas are used for database high availability

- For an application to be highly available, its database must also be highly available.
- Deploy two databases in different zones:
  - All access is routed to the master.
  - The master synchronizes its data with the failover replica.
  - In the event of an outage, the failover becomes the master.



Google Cloud

For your applications to be highly available, your databases must be too. Failover replicas are used to ensure database reliability. Deploy two databases in different zones. One is the master and one is the failover. Requests go to the master. All data is synchronized with the failover. If the master ever goes down, the failover takes requests until the master is back up and running.



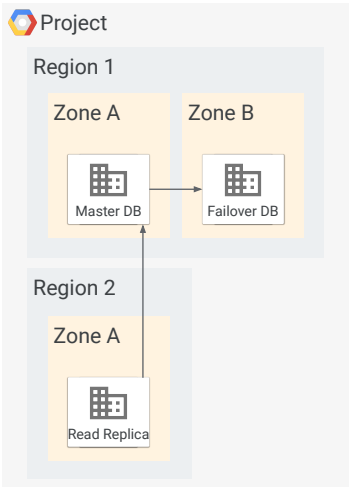
Scalable databases continue to work as the number of users and amount of data grow very large.

- To handle a **large volume of writes**, you split the database into pieces called shards. Then use multiple nodes, or servers, to process different shards.

- You can process **high volumes of reads** by creating multiple copies of the database called *replicas*. The read replicas handle analytics and reporting uses cases, while the master is left to handle the writes. The master ensures that the data is synchronized with the read replicas when changes are made.

For global applications, create read replicas in multiple regions

- One or more read replicas can be added for very large applications:
  - The master is responsible for the data.
  - It synchronizes the data with the replicas.
  - The replicas run the SELECT queries.
- For global applications, replicate across regions for lower latency to customers:
  - Beware of latency when replicating.
  - Strong vs. eventual consistency



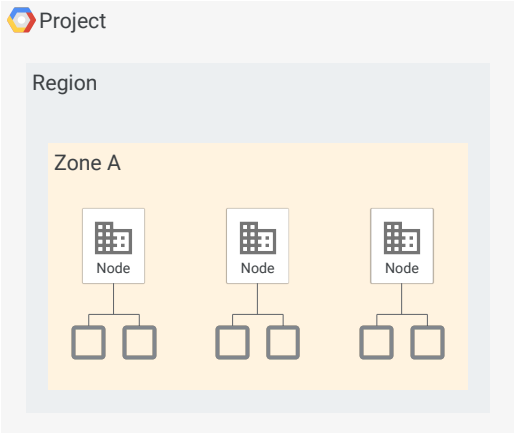
Google Cloud

Customers with users all over the world can create read replicas in multiple regions. Requests from users are routed to the region geographically closest to them. This can be automated using Google's global load balancers.

A master will still handle the writes and synchronize those changes to the replicas. When synchronizing across regions, you just have to be aware of the increased latency around the world. If you are using asynchronous replication across regions, the replicas can have stale data for a short period of time. This is known as *eventual consistency,* as opposed to *immediate consistency*.

Distributed databases use clusters of nodes and multiple shards to process high volume data

- When writes happen so quickly that a single server can't keep up, split the data into pieces (*shards*):
  - Clusters of servers consist of nodes.
  - Each node is responsible for some of the shards.
- For high availability or low latency, replicate clusters across multiple zones or regions.



Google Cloud

Distributed databases use clusters of nodes and multiple shards to process high-volume writes. This diagram is overly simplified, but it gets the idea across. If you split the data into smaller pieces, you can use multiple servers to distribute the workload. As the workload increases, you can keep adding more nodes and more shards, thus providing a system that seems to be infinitely scalable.

This is known as *horizontal scaling* or *scaling out*, as opposed to *vertical scaling* or *scaling up,* where you make the database server bigger to handle greater workloads. Cloud Spanner is a relational database that scales horizontally by adding nodes. Cloud SQL databases scale vertically increasing memory, vCPUs, and disk space.