# Select the right cloud-based DB for your project

How to Choose the Best Database for Scalability, Consistency, and Performance

by Marian Veteanu

# Cloud database deployment models

In this presentation, we will analyze the options that cloud computing offers when it comes to databases. If you're planning to build a SaaS application or even perform a basic lift-and-shift of your app from on-premises to the cloud, you will almost certainly need to select a database.

At a high level, there are three main ways to deploy your database:

**Traditional Database in a VM (IaaS):** You run a traditional database system (like MySQL, PostgreSQL, or SQL Server) on a virtual machine (VM) in the cloud. You manage everything, including patching, backups, scaling, and monitoring. You're probably considering this as a first step when doing a lift-and-shift operation.

Managed Cloud Database

**Database as a Service (DBaaS):** A database service offered by all major cloud providers where they manages most operational tasks, such as backups, patching, and scaling, while you maintain control over configurations and performance settings.

**Serverless DB:** A fully managed, database model that automatically scales based on workload demands. Users are charged based on actual usage, with no need to provision capacity in advance. You interact only with the database for querying and data management.

**Managed Cloud Database** is a broad term that refers to any database where the cloud provider handles the majority of operational tasks, such as backups, scaling, patching, and updates. Both **DBaaS (Database as a Service)** and **Serverless Database** fall under this umbrella because they are managed by the cloud provider, but they differ in the level of automation and scaling.

# Single node or Distributed

Should you deploy a **single-node** database or a **distributed** DB?

## Single Node Databases

A database that runs on a single server, with all data stored, processed, and queried on that one machine.

**Best for** small to medium applications with **predictable traffic** and applications that do not require global availability or high availability across regions.

**Examples**: MySQL, PostgreSQL, SQLite.

- **Simplicity**: Easier to manage and configure, suitable for smaller-scale applications.
- **Low Latency**: Faster access to data since it's all local to one machine.
- **Limited Scalability**: Vertical scaling is required (adding more CPU, RAM, or storage to a single machine). However, this approach has limitations as the hardware eventually maxes out.
- **Risk of Single Point of Failure**: If the server goes down, the entire database becomes unavailable, unless there's a backup and failover system.

## Distributed Databases

A DB that spans across multiple servers (or nodes), where data is distributed across the nodes in a cluster. These nodes can be located in different data centers or geographic regions.

**Best for** large-scale applications requiring **high availability** and **global distribution** and applications with **unpredictable or rapidly growing traffic** and **high read/write throughput**.

**Examples**: Amazon DynamoDB, Google Cloud Spanner, Apache Cassandra.

- **Horizontal Scalability**: Easily add more nodes to handle increased data and workload demands, providing nearly unlimited scaling.
- **Fault Tolerance**: Built-in **HA** and redundancy. If one node fails, another can take over.
- **Global Availability**: Data can be replicated across multiple regions, ensuring low-latency access for users across the globe.
- **Complexity**: More complex to manage and operate.

# Overview of Database Types

Understanding the different database types is crucial. Each type is designed to meet specific use cases based on data structure, scalability, and application requirements.

## Relational Databases (SQL)

Relational databases store data in structured tables using SQL (Structured Query Language) for querying and managing data.

They feature ACID compliance for strong consistency and reliability and are best for transactional applications with complex queries and relationships between data (e.g., joins).

Examples: MySQL, PostgreSQL, Microsoft SQL Server

## NewSQL Databases

Combine the scalability of NoSQL with the ACID compliance and query capabilities of traditional relational databases, to support large-scale, distributed transactions while maintaining strong consistency.

Examples: Google Spanner, CockroachDB

## NoSQL Databases

NoSQL databases are designed for unstructured, schema-less data, offering scalability across distributed systems.

They work best for distributed applications like real-time data processing, IoT systems, and social networks.

Examples:

- **Document-Based:** Store data in flexible, JSON-like documents (e.g. MongoDB, Couchbase).
- **Key-Value Store:** Simple key-value pairs, optimized for fast lookups (e.g. Amazon DynamoDB, Redis).
- **Column-Family:** Data stored in columns rather than rows, for high-scale analytics (e.g. Cassandra).

# Programming SQL vs NoSQL databases

**SQL** and **NoSQL** databases differ in how they store and query data. SQL databases use structured tables with relationships, while NoSQL databases use flexible, schema-less formats like documents, key-value pairs, or graphs. The programming experience differ between these two types of DBs.

### MySQL example

```
try {
    const sql = 'SELECT * FROM users WHERE age > ?';
    const [rows] = await connection.execute(sql, [25]);

    console.log(rows);
} catch (error) {
    console.error('Error executing query:', error);
} finally {
    await connection.end();
}
```

### Firebase (a NoSQL DB) example

```
const db = admin.firestore();

const usersRef = db.collection('users');
const query = usersRef.where('age', '>', 25);

try {
    const snapshot = await query.get();
    if (snapshot.empty)
        return;

    snapshot.forEach(doc => {
        console.log(doc.id, '=>', doc.data());
    });
} catch (error) {
    console.error('Error fetching documents:', error);
}
```

# About Data Consistency

Data consistency refers to the correctness and uniformity of data across all nodes or systems and impacts how and when data becomes available across distributed systems.

## Strong Consistency

Data is always up-to-date across all nodes. The DB guarantees that after a transaction is committed, all subsequent reads will reflect that committed data across all replicas.

Select a DB with strong consistency if you're building an app where data integrity is critical (e.g. banking, financial transaction, enterprise app, etc.)

**Trade-offs:** You Can expect latency due to coordination between replicas to ensure data consistency before responding to requests.

**What DB?:** Almost all relational databases (SQL based) offer strong consistency. Some new distributed systems like Google Cloud Spanner also favor strong data consistency.

## Eventual Consistency

Ensures that, given enough time, all replicas will eventually reflect the most recent write, but there is no immediate guarantee of consistency across nodes after a transaction.

DBs in this category, prioritizes availability over strict consistency, allowing for faster performance and lower latency.

Select this type of DB for distributed applications with high availability needs and tolerable inconsistencies (e.g., a social network).

**Trade-offs:** There is a risk of reading stale data immediately after a write, but the system is highly available.

**What DB?:** Most NoSQL databases, Apache Cassandra.

The **CAP Theorem** highlights the trade-off between **Consistency**, **Availability**, and **Partition** (e.g. network comm.) Tolerance. **Strong consistency** sacrifices availability or performance for accurate data. **Eventual consistency** prioritizes availability and performance but allows temporary inconsistencies. Consistency and Availability is unrealistic in distributed systems.

# It's Physics: The CAP Theorem

The **CAP Theorem** (Consistency, Availability, and Partition Tolerance), introduced by Eric Brewer, defines fundamental limits for distributed systems.

Much like physical laws that govern the natural world, CAP constrains what distributed databases can achieve when a network failure (partition) occurs.

**Consistency (C)**: Every read receives the most recent write or an error, ensuring up-to-date data across all nodes.

**Availability (A)**: Every request receives a response, even if it returns outdated data during network issues.

**Partition Tolerance (P)**: The system continues operating despite a network partition, where some nodes cannot communicate.

Just as you can't break the laws of physics, a distributed DB can only have two of the three properties during a network partition:

**Consistency and Partition Tolerance (CP):** The system ensures consistency but may sacrifice availability. E.g. Google Cloud Spanner

**Availability and Partition Tolerance (AP):** The system remains available but might show inconsistent data. E.g. Amazon DynamoDB

# Why Eventual Consistency Databases Scale Better?

Eventual consistency sacrifices immediate consistency in favor of increased **write performance** and **lower latency**.

The performance boost and reduced latency allow eventual consistency databases to handle **high-volume**, **low-latency** workloads with better efficiency, particularly for applications that don't require instant consistency.

This design choice provides several key advantages in scalability:

- In eventual consistency systems, data is replicated asynchronously across multiple nodes or regions, allowing write operations to be processed locally without waiting for synchronization.

- By not requiring all replicas to have the most up-to-date data at all times, eventual consistency systems allow nodes to operate more independently.

- Eventual consistency databases are designed to tolerate network partitions, meaning they continue to function even if nodes in the system become temporarily disconnected.

- With eventual consistency, read and write operations can be distributed across multiple nodes without the need for real-time coordination, reducing the load on individual servers.

# Strong Consistency Databases

Strong consistency ensures that every read operation returns the most recent write, guaranteeing that all users and applications see the same data, regardless of which node in a distributed system they access.

Strong consistency is typically associated with ACID (Atomicity, Consistency, Isolation, Durability) guarantees, which are critical for transactional integrity.

**Use Cases for Strong Consistency**

Most enterprise applications, healthcare systems, banking and financial applications, etc.

**DBs with strong consistency**

**Relational databases**

- Oracle Database
- Microsoft SQL Server
- IBM Db2
- MariaDB
- PostgreSQL
- MySQL (with InnoDB Engine)

**Distributed databases**

- Google Cloud Spanner
- CockroachDB
- Amazon Aurora
- Azure Cosmos DB (when strong consistency level selected)

# High Availability Databases

In databases, HA is achieved through redundancy, failover mechanisms, and distributed architectures, ensuring continuous service even in the event of hardware, software, or network failures.

### NoSQL Databases using Eventual Consistency & High Availability

**Amazon DynamoDB**: Fully managed NoSQL database offering automatic replication across multiple AWS regions, ensuring high availability and fault tolerance.
**Azure Cosmos DB** (certain modes)**:** Cosmos DB provides multiple consistency models, including eventual consistency. This is the lowest-latency option, which ensures that all replicas eventually converge to the same state.
**Apache Cassandra**: Distributed NoSQL database with peer-to-peer architecture, enabling multi-node replication for HA and scalability.

### NewSQL Databases with Strong Consistency & High Availability

**Google Cloud Spanner**: A globally distributed, strongly consistent database that ensures HA through synchronous replication and automatic failover across regions.
**CockroachDB**: A NewSQL database designed for high availability, with automatic replication, failover, and horizontal scaling across nodes and regions.

### Managed SQL Databases configured for High Availability*

**Amazon RDS**: Offers high availability with Multi-AZ (Availability Zone) deployments, providing automatic failover in case of instance failure.
**Azure SQL Database**: Managed relational database with built-in HA through active geo-replication, allowing continuous service even during regional failures.
**Google Cloud SQL**: Fully managed relational database service with automatic backups, replication, and HA through failover across zones and regions.

* Traditional SQL databases are generally better suited for vertical scaling rather than horizontal scaling (distributing data across multiple nodes). This can limit the scalability of SQL databases configured for HA when handling very large-scale workloads.

# Serverless Databases

**Serverless Databases** offer a cloud-native, fully managed solution that automatically scales with demand and charges only for the resources used.

Serverless databases can be designed to prioritize either **strong consistency** or **high availability**, depending on the application's needs.



**Amazon Aurora Serverless**

Relational database (MySQL and PostgreSQL compatible) that automatically adjusts capacity based on demand.

**Azure SQL DB Serverless**

Auto-pauses during inactivity and scales resources automatically, providing cost-efficient database solutions for low-traffic applications.

**Google Cloud Firestore**

A fully managed NoSQL document database that scales automatically with traffic, ideal for mobile, web, and serverless applications.

# Use case-based database selection

**Transactional Applications (OLTP)**

**Best Fit:** Relational Databases (SQL)
**Examples:** E-commerce platforms, banking systems, ERP systems.
**Recommended Databases:**

- Google Cloud SQL (MySQL/ PostgreSQL/ SQL Server)
- Amazon RDS
- Azure SQL Database

**Why:** Supports ACID transactions, complex queries, and strong consistency.

**Real-Time Analytics**

**Best Fit:** NoSQL or Column-Family Databases
**Examples:** IoT applications, sensor data analytics, real-time dashboards.
**Recommended Databases:**

- Google Cloud Bigtable
- Amazon DynamoDB
- Azure Cosmos DB

**Why:** Designed for high throughput, low-latency reads and writes, and horizontal scalability.

**Content Management and Document Storage**

**Best Fit:** Document Databases
**Examples:** Blogs, content-heavy websites, CMS platforms.
**Recommended Databases:**

- MongoDB Atlas
- Azure Cosmos DB (MongoDB API)
- Google Firestore

**Why:** Flexible schema design with JSON-based documens as the data model may vary between records.

**Scalable Global Applications**

**Best Fit:** Globally Distributed Databases
**Examples:** Global e-commerce platforms, SaaS platforms.
**Recommended Databases:**

- Google Cloud Spanner
- Amazon Aurora Global Database
- Azure Cosmos DB

**Why:** Provides global replication with low-latency access and strong consistency.

# Popular Databases in AWS - 1

## Relational Databases (SQL)

### Amazon RDS

- Supports popular engines like MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server.
- Managed database with automatic backups, scaling, patching, and high availability through Multi-AZ deployments.

### Amazon Aurora

- A MySQL- and PostgreSQL-compatible relational database designed for performance and availability.
- Offers up to 5x the throughput of standard MySQL and 3x that of PostgreSQL with automatic scaling and failover across multiple availability zones.
- **Amazon Aurora** offers a **serverless** option called **Aurora Serverless**, which is a fully managed, on-demand, auto-scaling configuration for Aurora.

| Feature | Amazon RDS | Amazon Aurora |
|---|---|---|
| **Supported Engines** | MySQL, PostgreSQL, MariaDB, Oracle, SQL Server | MySQL-compatible, PostgreSQL-compatible |
| **Performance** | Standard (comparable to traditional RDBMS) | 5x MySQL and 3x PostgreSQL performance |
| **Storage Scaling** | Manual scaling, up to 64 TB | Auto-scaling, up to 128 TB |
| **High Availability** | Multi-AZ support (optional) | Built-in HA with 6 copies across 3 AZs |
| **Global Replication** | No built-in global replication | Aurora Global Database with multi-region replication |
| **Read Replicas** | Available, up to 5 replicas | Up to 15 low-latency read replicas |

# Popular Databases in AWS - 2

## NoSQL Databases

**Amazon DynamoDB**
- Fully managed, key-value, and document NoSQL database.
- Supports automatic scaling, multi-region replication, and built-in high availability for large-scale applications.

**Amazon DocumentDB (with MongoDB compatibility)**
- Fully managed document database service that supports MongoDB workloads.
- Designed for scalability, availability, and fully managed cluster replication.

**Amazon ElastiCache**
- Managed in-memory data store supporting Redis and Memcached.
- Provides sub-millisecond latency for real-time applications like caching, session storage, and analytics.

**Amazon Neptune**
- Fully managed graph database service that supports both RDF and property graph models.

| Feature | Amazon DynamoDB | Amazon DocumentDB |
|---|---|---|
| **Database Type** | NoSQL (Key-Value, Document) | NoSQL (Document-based, MongoDB-compatible) |
| **Query Language** | DynamoDB API (NoSQL-specific queries) | MongoDB API (Supports MongoDB's query language) |
| **Use Case** | High-volume, low-latency applications, key-value | Applications requiring MongoDB compatibility |
| **Best for** | Key-value and simple document storage, highly scalable apps | MongoDB-based workloads needing scalability and management in AWS |

# Popular Databases in Azure - 1

## Relational Databases (SQL)

### Azure SQL Database

- Fully managed relational database service based on SQL Server.
- Supports automatic scaling, high availability, and built-in security.

### Azure SQL Managed Instance

- Managed instance of SQL Server with near 100% compatibility with the on-premises SQL Server.

### Azure Database for MySQL

- Managed MySQL database service with high availability and scalability.
- Ideal for open-source applications and developers familiar with MySQL.

### Azure Database for PostgreSQL

- Managed PostgreSQL service offering features like auto-scaling, backups, and high availability.
- Suitable for enterprise applications and analytics workloads.

| Feature | Azure SQL Database | Azure SQL Managed Instance |
|---|---|---|
| **Compatibility** | Designed for cloud-native applications with limited SQL Server compatibility | Near 100% compatibility with on-premises SQL Server, ideal for migrations |
| **Scalability** | Supports horizontal scaling through elastic pools | Vertical scaling within the instance; no support for elastic pools |
| **High Availability** | Built-in high availability with geo-redundancy and automatic failover | Built-in high availability with automatic failover within the VNet, supports multiple availability zones |
| **Use Case** | Best for new cloud-native apps or modernizing existing apps | Ideal for lift-and-shift migrations of on-prem SQL Server to the cloud |

# Popular Databases in Azure - 2

## NoSQL Databases

### Azure Table Storage

- NoSQL key-value store for structured data.
- Optimized for fast access and high availability at a low cost.

---

### Azure Cosmos DB

- Globally distributed, multi-model database that supports document, key-value, graph, and column-family data models.
- Offers multiple consistency models and is ideal for high-availability, low-latency applications.
- **Azure Cosmos DB** is a multi-model database service that supports several APIs*, allowing developers to work with different data models and query languages.

**SQL API:** The default and most widely used API in Cosmos DB. Ideal for applications that require querying and manipulating JSON documents using familiar SQL commands.

```
SELECT * FROM c WHERE c.age > 30
```

**MongoDB API:** Ideal for MongoDB developers who want to benefit from Cosmos DB's global distribution and scalability without changing their MongoDB codebase.

```
db.collection.find({ "age": { $gt: 30 } })
```

**Cassandra API:** Allows applications built for Cassandra to run on Cosmos DB using CQL (Cassandra Query Language).

Ideal for applications that need high throughput and low-latency access to distributed, wide-column data.

```
SELECT * FROM users WHERE age > 30;
```

**Gremlin API:** Supports the Gremlin graph traversal language

Nodes (vertices) and edges (relationships) are used to model complex networks.

```
g.V().has('person', 'age',
gt(30)).values('name')
```

* Other API supported. Consult documentation.

# Popular Databases in GCP - 1

**Relational Databases (SQL)**

**Cloud SQL:** Fully managed relational database service for MySQL, PostgreSQL, and SQL Server.
Provides automated backups, replication, and high availability for SQL workloads.

---

**NewSQL Databases**

**Cloud Spanner:** Fully managed, globally distributed, and horizontally scalable NewSQL database. It combines the best aspects of traditional relational databases (SQL, ACID compliance) with the performance, scalability, and availability of NoSQL databases.

Supports standard SQL queries, including joins, indexes, and complex transactions, making it easy for developers familiar with traditional relational databases to adopt.

Spanner uses a relational database model with support for SQL queries, joins, and strong consistency with ACID transactions

Spanner provides horizontal scaling by sharding data across multiple nodes. It can handle growing workloads and datasets without performance degradation, unlike traditional relational databases that require vertical scaling.

Cloud Spanner is globally distributed, meaning it can automatically replicate data across multiple regions and ensure low-latency access anywhere in the world.

# Popular Databases in GCP - 2

**NoSQL Databases**

**Google Firestore (part of Firebase)**

- Fully managed NoSQL document database, ideal for mobile, web, and serverless applications.
- Supports real-time updates and offline data synchronization.

**Google Bigtable**

- Managed NoSQL wide-column store designed for large-scale, low-latency workloads like time-series data, IoT, and financial data.
- Provides seamless scalability with automatic sharding and replication.

**Google BigQuery**

- Fully managed, serverless, highly scalable data warehouse designed for large-scale data analytics.
- Provides lightning-fast SQL-based querying over massive datasets and supports integration with ML models for AI-powered analytics.

"Bigtable is a NoSQL wide-column database optimized for heavy reads and writes."

"BigQuery is an enterprise data warehouse for large amounts of relational structured data."

# Other Popular DBaaS Providers

Beyond the major cloud platforms like AWS, Azure, and Google Cloud, several other **Database as a Service (DBaaS)** providers offer specialized database solutions.

**MongoDB Atlas:** Fully managed cloud database for MongoDB, offering cross-cloud support on AWS, Azure, and Google Cloud.

**CockroachDB Cloud:** A fully managed distributed SQL database offering strong consistency, high availability, and horizontal scalability.

**Fauna:** A globally distributed serverless database that supports GraphQL and ACID transactions.

**PlanetScale:** A MySQL-compatible distributed database built on Vitess, designed for large-scale applications.

**LibSQL:** A cloud-hosted fork of SQLite that brings cloud-native features to the simplicity of SQLite.

# Key Considerations for Selecting a Cloud Database - 1

## Scalability

**Vertical vs. Horizontal Scaling:** Determine if the database can scale vertically (increasing compute resources) or horizontally (adding more instances/nodes).

**Auto-scaling Capabilities:** For simplified management, look for databases that automatically adjust resources based on demand, like serverless databases.

## Performance and Latency

**Low Latency Requirements:** Choose databases that provide fast response times, especially for real-time applications.

**Geographical Distribution:** If your users are distributed globally, ensure the database offers multi-region deployments to minimize latency.

## Cost Efficiency

**Pay-as-You-Go vs. Reserved Capacity:** Analyze your workload to decide between on-demand pricing or reserved instances for long-term savings.

## Data Consistency vs. Availability

**ACID Compliance:** For transactional systems that require strong consistency, choose a relational database.

**Eventual Consistency:** NoSQL databases often trade strong consistency for better availability and partition tolerance, suitable for distributed apps.

# Key Considerations for Selecting a Cloud Database - 2

## Security and Compliance

**Data Encryption:** Ensure the database offers encryption at rest and in transit

**Identity and Access Management (IAM):** Look for integrated IAM controls

**Compliance Requirements:** GDPR, HIPAA, or SOC 2

## High Availability and Disaster Recovery

**Built-in Redundancy:** Check if the database offers automated failover, replication, and backup features.

## Integration with Other Services

**Cloud Ecosystem:** Ensure the database integrates well with other services (e.g., analytics, machine learning, data lakes) in your chosen cloud provider's ecosystem.

**Third-Party Tools:** Check compatibility with external tools for data pipelines, business intelligence, or monitoring.

## Vendor Lock-In

**Cross-Platform Support:** Consider whether you need the flexibility to migrate the database across different cloud providers or hybrid environments.

**Open-Source Options:** If avoiding vendor lock-in is critical, explore open-source databases that can be self-managed or used across platforms.

**Marian Veteanu**

Technology Architect and Product Leader

Looking to see how I can
add value to your organization?

Message me!

https://www.linkedin.com/in/mveteanu/
https://x.com/mveteanu