AZURE COSMOS DB THEORETICAL Q&A

BY - SHUBHAM WADEKAR

1. What is Azure Cosmos DB?

Azure Cosmos DB is a cloud-based NoSQL database service offered by Microsoft. It is designed to store and process large volumes of unstructured and semi-structured data. Cosmos DB is fully managed, which means Microsoft handles the infrastructure, backups, and scaling, so I can just focus on building my application.

It supports multiple data models like key-value, document, graph, and column-family, so I can use it for many different types of applications. One of the main features I like is that it is globally distributed, which means I can replicate my data across different regions to improve performance and availability. It also offers very fast response times, often in less than 10 milliseconds for reads and writes.

2. What is a NoSQL database, and how does it work?

A NoSQL database is a type of database that stores data in formats other than traditional tables used in relational databases. Instead of using rows and columns, NoSQL databases use flexible formats like JSON documents, key-value pairs, wide-column stores, or graph structures.

These databases are designed to handle large volumes of data and are especially useful when the data structure can change over time or when we need high speed and scalability. They allow me to store and access data without requiring a fixed schema, which is very helpful for modern applications like mobile apps or real-time analytics systems.

In NoSQL databases, data is usually grouped based on how it is accessed, which makes queries faster. Also, they scale out horizontally, meaning I can add more servers to handle more traffic, unlike relational databases which usually scale up by increasing server capacity.

3. What distinguishes Cosmos DB from other databases?

Cosmos DB stands out because it offers multiple unique features that are not commonly found together in other databases.

First, it supports five different APIs, including SQL for querying JSON documents, MongoDB API, Cassandra API, Gremlin for graph data, and Table API. This means I can use Cosmos DB in many different ways, depending on the needs of my application, without changing the underlying service.

Second, it offers guaranteed low latency of less than 10 milliseconds for reads and writes at the 99th percentile. This is very important for applications that need real-time performance.

Third, Cosmos DB is globally distributed by default. I can replicate my data to any Azure region with just a few clicks, and this helps users from different parts of the world get faster access to data.

Fourth, it offers a unique feature called multi-master replication, which allows read and write operations to be performed in multiple regions at the same time. This improves both availability and performance.

Finally, Cosmos DB provides automatic and instant scalability based on workload, and I can also choose between provisioned throughput and serverless models based on my cost and performance requirements. All of these features combined make Cosmos DB a strong choice for cloud-native, highly responsive, and globally available applications.

4. How does Azure Cosmos DB differ from traditional relational databases?

Azure Cosmos DB is a NoSQL database, while traditional relational databases like SQL Server or Oracle use structured tables with fixed schemas. In Cosmos DB, the data is usually stored in flexible formats like JSON documents, which means the structure of the data can change over time without any issues. This is very useful when the application evolves frequently or deals with different types of data.

Relational databases use SQL for querying and rely heavily on joins and complex relationships between tables. On the other hand, Cosmos DB is optimized for fast reads and writes without the need for joins. It stores data in a way that is more aligned with how modern applications access it.

Another major difference is in scaling. Relational databases usually scale vertically, which means upgrading the server to handle more load. Cosmos DB scales horizontally, which means adding more machines to handle increased traffic, and this makes it better suited for large-scale web or mobile applications.

Also, Cosmos DB is globally distributed by design, and it can replicate data across multiple regions automatically. Relational databases usually need extra setup and effort to support that kind of distribution.

5. What is the difference between Azure Cosmos DB and Azure DB?

Azure Cosmos DB and Azure SQL Database (which is often referred to as Azure DB) are two different database services designed for different use cases.

Azure Cosmos DB is a NoSQL database. It is used for unstructured or semi-structured data like JSON. It supports multiple data models and is great for applications that need high speed, global access, and flexibility in data structure.

Azure SQL Database is a fully managed version of the SQL Server database in the cloud. It uses traditional relational models with tables, rows, and columns. It is best suited for applications that need strong relationships between data and use structured schemas.

In short, if I need to handle fast-growing, flexible data like sensor data, user activity, or mobile app content, Cosmos DB is a better choice. If I need structured data with transactions and relationships, like billing systems or accounting, then Azure SQL Database is more suitable.

6. What is a JSON document defined as?

A JSON document is a way to store data using key and value pairs. It stands for JavaScript Object Notation. It is a lightweight format that is easy for both humans and machines to read and write. In a JSON document, data is represented using curly braces, and each item is stored as a field name followed by its value.

For example, a JSON document storing user information might look like this:

```
{
  "id": "user123",
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com"
}
```

Each piece of data is stored as a field, and the value can be a number, string, boolean, array, or even another nested JSON object. Cosmos DB uses JSON documents to store data because they are flexible and allow different documents to have different fields, which is not possible in traditional tables. This flexibility makes it easier to work with changing or unstructured data.

7. What is the definition of a primary key?

A primary key is a unique identifier for each item or record in a database. It makes sure that every piece of data in the collection or table can be uniquely identified. In Azure Cosmos DB, when we create a container, we define a partition key, but each document also needs a unique identifier called the id property. Together, the combination of the partition key and id forms the primary key.

This is important because it allows the database to quickly find, update, or delete a specific document without any confusion. The primary key must be unique across the container so that no two documents have the same key combination.

8. What are the different API types used in Azure Cosmos DB?

Azure Cosmos DB supports five main types of APIs, each designed for different types of data models and application needs:

1. SQL API: This is the default and most commonly used API. It lets me use SQL-like queries to access and manage JSON documents in the database.
2. MongoDB API: This allows Cosmos DB to act like a MongoDB database. It is useful if I have existing applications that already work with MongoDB and want to move to Cosmos DB without changing the code much.
3. Cassandra API: This lets me use Cassandra query language to work with wide-column data. It is good for applications that use large volumes of data with high write throughput.
4. Gremlin API: This is used for graph databases where data is stored as nodes and relationships. It is helpful for things like social networks or recommendation systems.
5. Table API: This supports applications that use Azure Table storage. It is a simple key-value store, mainly for lightweight data access scenarios.

9. Which API works well with Cosmos DB?

The answer depends on what the application needs, but in general, the SQL API works best with Cosmos DB. It is the most fully supported API and offers the most features. It is designed specifically for Cosmos DB and supports rich querying on JSON documents, indexing, and built-in functions. I usually choose the SQL API when building a new application from scratch because it gives me the most control and performance with Cosmos DB.

However, if I am migrating an existing application from MongoDB or Cassandra, then using the MongoDB API or Cassandra API can be a better fit since I won’t need to change the application code much. Each API works well for the purpose it is built for, but if I am starting fresh and want to take full advantage of Cosmos DB’s features, I prefer to go with the SQL API.

10. Can I use multiple APIs to access my data?

No, in Azure Cosmos DB, I cannot use multiple APIs to access the same data within a single container or database. When I create a Cosmos DB account, I have to choose one API type, such as SQL API, MongoDB API, Cassandra API, Gremlin API, or Table API. That choice determines how the data is stored and how it will be accessed.

Each API uses a different wire protocol and storage format behind the scenes. So, for example, if I create a Cosmos DB account using the SQL API, I cannot later use the MongoDB API to query the same data in that account. If I want to use a different API, I would need to create a separate Cosmos DB account and move or sync the data into that new account using the preferred API.

11. What is the SQL API for Azure Cosmos DB?

The SQL API is the default and most popular API provided by Azure Cosmos DB. It allows me to store and query data in the form of JSON documents using a language that is very similar to traditional SQL.

When I use the SQL API, I can perform queries using SELECT statements, filter with WHERE clauses, and even use built-in functions to manipulate or search the data. This API is optimized specifically for document-based data and is the most feature-rich option in Cosmos DB.

It also provides support for automatic indexing, stored procedures, triggers, and user-defined functions. That means I can build powerful and flexible applications directly on top of the database without needing to do much manual configuration.

12. What does Azure Cosmos DB's SQL API do?

The SQL API in Azure Cosmos DB allows me to interact with data stored in JSON format using a familiar SQL-like language. It helps me perform a wide range of actions, such as:

- Inserting new JSON documents into a container
- Querying documents using SELECT statements
- Filtering data using WHERE clauses
- Sorting, grouping, and projecting fields
- Updating or deleting documents
- Creating stored procedures, triggers, and user-defined functions for advanced logic

What makes the SQL API very useful is that it supports rich indexing by default. That means I don't have to manually define indexes unless I want to customize performance. It can handle nested JSON structures, arrays, and complex queries very efficiently.

For example, if I have a container storing customer orders, I can run a query like:

SELECT c.customerName, c.orderTotal

FROM c

WHERE c.orderTotal > 500

This makes it easy to access and analyze data in real time, which is a key requirement in many modern applications.

13. What are the components of the SQL API for Azure Cosmos DB?

The SQL API in Azure Cosmos DB is made up of several important components that help in storing, querying, and managing JSON data. These components include:

1. **Containers**: These are like tables in a traditional database. A container stores the JSON documents and is the main storage unit in Cosmos DB.
2. **Items (Documents)**: These are the actual JSON objects stored in the container. Each item represents a single data record.
3. **Partition Key**: This is a value inside the JSON document that is used to distribute the data across multiple partitions for scalability and performance.
4. **Indexing Engine**: Cosmos DB automatically indexes all data stored in containers. This helps in fast query performance without needing to create indexes manually.
5. **SQL Query Language**: This is the language used to query the JSON data. It looks like traditional SQL but is designed to work with document-based data.
6. **Stored Procedures, Triggers, and User-Defined Functions (UDFs)**: These are written in JavaScript and run within the database to handle business logic close to the data.
7. **Resource Model**: Every component, like databases, containers, and items, is treated as a resource with a unique identifier.
8. **Request Units (RUs)**: This is how performance is measured. Each operation, like reads or writes, consumes RUs based on the complexity and size of the operation.

These components work together to provide a powerful and flexible system for building applications that need fast and scalable data access.

14. When should you utilize the SQL API for Azure Cosmos DB?

I should use the SQL API when I am working with JSON data and need a fully managed, high-performance NoSQL database. It is a great fit in the following cases:

- When I am building modern web, mobile, or IoT applications that need fast reads and writes
- When the structure of my data can change frequently or is semi-structured
- When I need to scale globally and serve users in different regions
- When I want to use a familiar query language similar to SQL to work with document data
- When I want to use advanced features like stored procedures and triggers for data processing
- When I need automatic indexing and do not want to manage database internals
- When I want to build real-time dashboards or reporting systems that need fast filtering and aggregations

In simple words, I use the SQL API when I want flexibility, speed, and a familiar way to work with document-based data.

15. What is the Azure Cosmos DB API for MongoDB?

The Azure Cosmos DB API for MongoDB allows me to use Cosmos DB as if it were a MongoDB database. This means I can connect my MongoDB applications to Cosmos DB without changing the existing MongoDB client code, drivers, or tools.

Behind the scenes, Cosmos DB will store the data using its own storage engine, but it will understand and respond to MongoDB commands and queries. This is helpful when I want to migrate a MongoDB-based application to the cloud and take advantage of Cosmos DB's features like global distribution, automatic scaling, high availability, and low latency.

With this API, I can use collections, documents, and MongoDB's query language, just like I would in a regular MongoDB setup. It is especially useful if my development team already knows MongoDB and I want to move to a fully managed cloud solution without rewriting the whole application.

16. What are the different data models supported by Azure Cosmos DB?

Azure Cosmos DB supports five different data models, which allows it to handle a wide range of application needs. These data models are:

1. document model
2. key-value model
3. column-family model
4. graph model
5. table model

Each model is supported through a specific API. For example, the document model is supported by the SQL API and MongoDB API, the key-value model is supported by the Table API, the column-family model is supported by the Cassandra API, and the graph model is supported by the Gremlin API.

This flexibility allows me to use Cosmos DB for different types of applications, like storing user profiles, managing sensor data, handling social network relationships, or building recommendation engines.

17. How does Azure Cosmos DB support various data models such as key/value, columnar, document, and graph?

Azure Cosmos DB is designed with a multi-model architecture. This means that even though the underlying storage engine is the same, it can present the data in different formats depending on the API I choose. Here is how it supports each model:

- for key/value data, Cosmos DB provides the Table API. It stores data as key-value pairs where each item is accessed using a unique key.
- for column-family data, Cosmos DB uses the Cassandra API. This allows storing data in rows and columns, like in a wide-column database. Each row can have a different set of columns.
- for document data, Cosmos DB uses the SQL API and MongoDB API. It stores data in JSON format, where each document can have a flexible structure.

- for graph data, Cosmos DB provides the Gremlin API. This is used to model relationships and connections between items using vertices and edges.

Each API is designed to make Cosmos DB behave like the corresponding database system, so I can use familiar tools and query languages. The actual data is stored in a single backend system, but the way I access it depends on the API and model I choose.

18. What is an Azure Cosmos DB container?

A container in Azure Cosmos DB is the main storage unit where my data is kept. It stores items, which are usually JSON documents. I can think of a container like a table in a relational database, but it is more flexible because the documents inside do not need to follow a fixed schema.

Each container has the following properties:

- it can store millions of items
- it has a partition key to help distribute data across multiple servers for performance and scalability
- it supports automatic indexing, so I can query the data quickly
- it has a set amount of throughput, measured in request units, that determines how many operations it can handle per second

When I create a container, I decide the partition key and the throughput, and Cosmos DB takes care of everything else. Containers make it easy to manage, scale, and query large amounts of data in a fast and efficient way.

19. How can a new container be created?

A new container in Azure Cosmos DB can be created using the Azure portal, the Azure CLI, PowerShell, or through code using SDKs. The easiest way is through the Azure portal. Here's how I usually do it:

1. I go to the Azure portal and open my Cosmos DB account.
2. Then I click on the option called Data Explorer.
3. Inside Data Explorer, I click on the New Container button.
4. I enter the name of the database (or create a new one), the container name, and the partition key.
5. I also set the throughput either as fixed (manual) or serverless.
6. Finally, I click OK and the container is created.

Once the container is created, I can start adding items, running queries, and managing data

.

20. What's the best way to make a new container?

The best way to create a new container depends on the situation. If I am setting things up manually or doing quick testing, using the Azure portal is the fastest and easiest method because it has a user-friendly interface.

If I am automating deployments, managing infrastructure as code, or working with multiple environments, then using tools like the Azure CLI, ARM templates, or SDKs like the .NET or Python SDK is better. This helps me automate the process and keep everything consistent.

Also, when creating the container, I always make sure to choose the right partition key because that affects performance and scalability. A good partition key is one that has high cardinality and spreads the data evenly.

21. How can I create a new Azure Cosmos DB database?

Creating a new database in Azure Cosmos DB can be done through the Azure portal or programmatically. Here is how I usually create it through the portal:

1. I log in to the Azure portal and go to my Cosmos DB account.
2. In the left menu, I click on Data Explorer.
3. Then I click on the New Database button.
4. I enter a name for the new database.
5. If I want, I can assign a shared throughput to the database so all containers inside can use it.
6. After that, I click OK and the database is created.

Once the database is created, I can add containers to it. If I prefer to do this using code, I can also use the Cosmos DB SDK in languages like C#, Java, or Python to create the database with just a few lines of code. This is helpful when I am building applications or setting up resources automatically.

22. How to make a new database?

To make a new database in Azure Cosmos DB, I usually use the Azure portal because it's simple and doesn't require any coding. Here is the step-by-step process I follow:

1. I go to the Azure portal and open my Cosmos DB account.
2. Then I click on Data Explorer in the left-hand menu.
3. In Data Explorer, I click on the New Database option.
4. I give my database a unique name.
5. I choose whether I want to assign throughput at the database level (shared) or manage it separately for each container.
6. Finally, I click OK and the new database is created.

Once the database is created, I can start adding containers and storing data in it. I can also do this using command-line tools or code, but for most use cases, the portal is the easiest way to get started.

23. What is the procedure for creating a new account?

To create a new Azure Cosmos DB account, I follow these steps in the Azure portal:

1. I sign in to the Azure portal.
2. I click on Create a resource at the top-left corner.
3. I search for Azure Cosmos DB and select it.
4. I choose the API type I want to use, such as SQL, MongoDB, Cassandra, Gremlin, or Table.
5. Then I enter details like the subscription, resource group, account name, and region.
6. I also select options like multi-region writes, availability zones, and backup policies based on my needs.
7. Once everything is filled in, I click on Review + create and then Create.

After the deployment is complete, the new Cosmos DB account is ready, and I can start creating databases and containers inside it.

24. How to create a simple item?

To create a simple item, which is basically a JSON document, I use the Data Explorer in the Azure portal. Here is how I do it:

1. I go to the Azure portal and open my Cosmos DB account.
2. Then I click on Data Explorer.
3. I select the database and the container where I want to add the item.
4. I click on Items in the container view and then click on New Item.
5. In the editor that appears, I enter the JSON data. For example:

```
{
 "id": "1",
 "name": "John Doe",
 "age": 30,
 "city": "Mumbai"
}
```

6. I click Save, and the item is created and stored in the container.

Each item must have a unique id and must follow the partition key structure defined for the container. After the item is saved, I can query it, update it, or delete it using Data Explorer or from my application code.

25. How does Azure Cosmos DB offer predictable performance?

Azure Cosmos DB offers predictable performance by using a system called request units, or RUs. Every operation in Cosmos DB, like reading, writing, or querying data, consumes a certain number of RUs based on how much work it requires. For example, reading a small item uses fewer RUs than running a complex query.

When I set the throughput of my database or container, I specify how many RUs per second I want to reserve. Cosmos DB guarantees that those RUs will be available to my application at all times. This means my application will always have the performance it needs, even if there is a lot of traffic.

Cosmos DB also uses automatic indexing and a fast storage engine, so queries are optimized for speed. Since the performance is tied to the RUs I set, I can calculate and manage the performance ahead of time. This helps me avoid surprises and plan for both normal and high-traffic times.

26. What is Throughput in Cosmos DB?

Throughput in Cosmos DB is the measure of how much work the database can handle, and it is measured in request units per second. It represents the amount of resources I reserve for my operations. For example, I might set 400 RUs per second for a container, and Cosmos DB will guarantee that capacity is available all the time.

Every operation, like inserting a document or running a query, consumes RUs. The cost depends on factors like the size of the document, the complexity of the query, and whether an index is used. If my workload needs more performance, I can increase the throughput to avoid throttling.

I can assign throughput at either the container level or the database level. When I assign it to the database, all containers inside share the RUs. When I assign it to a container, only that container uses those RUs.

27. What are the throughput limits of Azure Cosmos DB?

Azure Cosmos DB is designed to handle massive scale, so it offers very high throughput limits. Some of the key points about throughput limits are:

- A single container can scale up to millions of request units per second.
- If I need even more, I can create multiple containers and distribute the load.
- The actual maximum depends on the partitioning. A well-designed partition key helps scale easily.
- For manual throughput, I can set a fixed number of RUs per second, like 400, 1000, or more.
- For serverless mode, I don’t reserve throughput. Instead, I pay for the total number of RUs consumed.

There is no hard upper limit from the platform for throughput, but practical limits depend on how well I design my partitioning and how much my workload grows. If needed, Cosmos DB support can help with increasing quotas for very high workloads.

28. How does Cosmos DB provide predictable results?

Cosmos DB provides predictable results by using a combination of strong consistency options, guaranteed low-latency reads and writes, and automatic indexing. It ensures that every query or operation returns accurate and consistent data, based on the consistency level I choose.

There are five consistency levels to choose from: strong, bounded staleness, session, consistent prefix, and eventual. I can pick the level that matches my application's need for accuracy and speed. For example, if I need strong consistency, Cosmos DB guarantees that all users see the most recent data, no matter where they are.

Cosmos DB also automatically indexes all data without requiring me to define indexes manually. This makes queries faster and more predictable. Combined with reserved throughput using request units, Cosmos DB ensures that queries and operations perform in a reliable and repeatable way, even during high traffic.

29. What is Cosmos DB's maximum size?

Cosmos DB is built to scale without hard limits, especially when data is distributed across partitions. For a single logical container, there is virtually no storage limit as long as the data is evenly spread across multiple partitions.

Each physical partition can store up to 50 gigabytes of data and handle up to 10,000 request units per second. But Cosmos DB can automatically create more partitions as needed. This means that a container can scale to petabytes of storage and millions of request units per second if I design my partition key properly.

So in practical terms, the database can grow as large as needed without needing to move to another system. That's one of the reasons it's used for high-scale applications like retail systems, IoT platforms, and online games.

30. Is Azure Cosmos DB fast?

Yes, Azure Cosmos DB is very fast. It is designed for low-latency performance and guarantees response times of less than 10 milliseconds for read and write operations at the 99th percentile. That means even under heavy load, most requests will finish very quickly.

The speed comes from several features:

- Data is automatically indexed, so queries return results quickly.
- Cosmos DB uses a distributed architecture, so data can be read and written in parallel across partitions.
- With multi-region writes enabled, I can write to the database from anywhere in the world and still get fast performance.
- Data is kept close to users by replicating it in multiple regions.

Because of these reasons, Cosmos DB is a great choice when I need high-speed access to data, such as in mobile apps, gaming platforms, telemetry systems, or e-commerce websites.

31. Explain the concept of partitioning in Azure Cosmos DB.

Partitioning in Azure Cosmos DB is a method used to divide data into smaller parts so it can be stored and processed more efficiently. When data grows large, it is not practical to keep everything in a single place. Partitioning helps spread the data across multiple servers, which improves performance and allows the system to scale easily.

In Cosmos DB, each container is split into logical partitions. Each partition is based on the value of a partition key, which is a specific field in the JSON documents. All documents that have the same value for the partition key are stored together in the same partition.

For example, if I use "country" as a partition key, all documents with the country set to "India" will go into one partition, and those with "USA" will go into another. Behind the scenes, these logical partitions are stored in physical partitions.

The main benefits of partitioning are:

- Better performance because data is accessed faster from smaller sections
- Automatic scaling because Cosmos DB can create more partitions as needed
- Efficient use of throughput because traffic is spread evenly

Choosing the right partition key is very important. A good key has many unique values and evenly spreads the data and traffic across partitions.

32. Explain how Cosmos DB indexes data.

Cosmos DB automatically indexes all the data that is stored in a container without requiring me to manually create or manage indexes. This indexing system is designed to support fast and efficient queries on JSON documents.

When I insert or update a document, Cosmos DB updates the index immediately in the background. This ensures that any queries I run later can quickly locate the data without scanning the entire container.

The index includes all properties of the document by default. This means I can run queries on any field, even nested ones, and still get good performance. Cosmos DB supports range queries, equality filters, sorting, and even queries on arrays using these indexes.

If I want more control, I can customize the indexing policy. I can exclude certain fields from being indexed to save storage and reduce write costs. I can also define how the data should be sorted and whether it should support range queries.

This automatic and flexible indexing system is one of the reasons why Cosmos DB can provide fast and predictable performance for complex queries.

33. What are the various levels of consistency that exist in Cosmos DB?

Azure Cosmos DB offers five levels of consistency to help balance between performance and data accuracy. These consistency levels control how up-to-date and accurate the data is when it's read from the database. The five levels are:

1. **Strong consistency**: This level guarantees the most recent data is always returned. It behaves like a traditional database where a read will always reflect the latest write. It offers the highest accuracy but has higher latency and limited availability across regions.
2. **Bounded staleness**: This level allows a small delay in reading the latest data. I can define either a time window or a number of versions (updates) that are acceptable. This is useful when I need almost up-to-date data with slightly better performance than strong consistency.
3. **Session consistency**: This is the default level. It guarantees strong consistency for a single user session. This means within my session, I will always read my own writes, but other users might not immediately see them. It provides a good balance between consistency and performance.
4. **Consistent prefix**: This level ensures that reads return data in the order it was written. It might not include the latest writes, but it never shows data out of order. It is useful when I want to maintain the sequence of operations.
5. **Eventual consistency**: This is the weakest level. It offers the best performance and availability, but data may not be immediately consistent across regions. Over time, all copies of the data will become the same, but immediate reads might show old data.

Choosing the right consistency level depends on the needs of the application. For example, banking apps may need strong consistency, while news feeds or telemetry systems may work fine with eventual consistency.

34. Are direct and gateway connectivity modes encrypted?

Yes, both direct and gateway connectivity modes in Azure Cosmos DB are encrypted using secure protocols. Whether my application connects using the direct mode or the gateway mode, all communication between the client and the Cosmos DB service is protected with encryption using Transport Layer Security, or TLS.

In direct mode, the client communicates directly with the backend nodes of Cosmos DB, and in gateway mode, the client connects through a gateway proxy. Even though the network paths are different, the data is still encrypted in both cases.

This means that sensitive data like user information or business records remains safe during transit and is not exposed to anyone on the network. Encryption in both modes is handled by Cosmos DB automatically, so I don’t need to set it up manually.

35. Is Azure Cosmos DB HIPAA compliant?

Yes, Azure Cosmos DB is HIPAA compliant. HIPAA stands for Health Insurance Portability and Accountability Act, and it sets rules for handling sensitive healthcare data. Cosmos DB meets the security and privacy requirements needed to store and process healthcare information.

This means that if I am building a healthcare application that deals with patient records or other medical data, I can use Cosmos DB as long as I also follow proper security practices in my application. Microsoft provides a Business Associate Agreement for HIPAA, which ensures that the platform is legally allowed to handle protected health data.

In addition to HIPAA, Cosmos DB is also compliant with many other standards like ISO, GDPR, SOC, and FedRAMP. This makes it suitable for use in industries where data protection and privacy are very important.

36. What are Authorization headers?

Authorization headers are a way to prove to Cosmos DB that a request is coming from an allowed and trusted source. Every time my application sends a request to Cosmos DB, it must include an Authorization header that contains a special token.

This token is generated based on the type of authentication I'm using, such as a master key, resource token, or Azure Active Directory token. The Authorization header tells Cosmos DB who I am and whether I have permission to perform the action, like reading or writing data.

Without a valid Authorization header, Cosmos DB will reject the request for security reasons. This is how the service protects access to the data and prevents unauthorized users from performing operations.

37. What is meant by Master Keys and how do they operate?

Master keys in Azure Cosmos DB are long, secret strings that are used to access the database account and perform operations. When I create a Cosmos DB account, Azure gives me two master keys: a primary key and a secondary key. These keys give full access to all databases, containers, and documents in the account.

I use these keys when I want to build a server-side application that needs full control over the database. The application includes the master key in the Authorization header to authenticate itself.

Because the master keys have full access, I need to keep them safe and secure. If a master key is accidentally shared or exposed, someone could read, write, or delete all the data in my account. That's why it's a good practice to regenerate the keys regularly and to avoid using them in client-side apps like mobile apps or web browsers.

Instead of using master keys in client apps, I can use resource tokens or Azure AD for more secure and limited access.

38. What are the storage limits of Azure Cosmos DB?

Azure Cosmos DB is designed to scale to very large amounts of data, and there is no fixed storage limit for a container or database as long as partitioning is used correctly. Each physical partition in Cosmos DB can store up to 50 gigabytes of data. However, if my data grows beyond that, Cosmos DB automatically adds more physical partitions in the background.

This means the overall storage is virtually unlimited. I can store terabytes or even petabytes of data by spreading it across multiple partitions using a good partition key. There is no need for me to manually manage this scaling—Cosmos DB handles it automatically.

So, the key point is: as long as I use a partition key that evenly distributes data, there is no practical limit to how much I can store.

39. How much does Azure Cosmos DB cost?

The cost of Azure Cosmos DB depends mainly on two things: the provisioned throughput and the amount of storage used.

Provisioned throughput is measured in request units per second. I can choose to use manual throughput (fixed RUs), autoscale throughput, or serverless.

- In fixed mode, I pay for the number of RUs I reserve, whether I use them or not.
- In autoscale, I pay based on the highest usage during a billing hour.
- In serverless, I pay only for what I use, which is good for small or unpredictable workloads.

Storage costs are based on the total amount of data and indexes stored in the database. This is charged per gigabyte each month.

There may be additional charges for features like multi-region replication, backups, or network usage. Microsoft provides a pricing calculator that helps estimate the monthly cost based on my workload and region.

40. What is meant by Azure Cosmos DB emulator?

The Azure Cosmos DB emulator is a tool I can install on my local machine to develop and test Cosmos DB applications without connecting to the cloud. It behaves just like the real Cosmos DB service, so I can write, read, query, and test my code using the same SDKs and APIs.

This is helpful because I don't need an active Azure subscription or pay for operations while I'm building and testing my app locally. Once I'm done with development, I can switch the connection string to point to the actual Cosmos DB account in the cloud.

The emulator supports features like the SQL API, local storage, authentication using keys, and even supports querying through the Data Explorer. It's only available for Windows, and it's mainly meant for development and testing—not for running production applications.

41. Is it possible to run DocumentDB API locally?

Yes, it is possible to run the DocumentDB API locally by using the Azure Cosmos DB emulator. The DocumentDB API was the original name for what is now known as the SQL API in Azure Cosmos DB. The emulator supports the SQL API, which means I can develop and test applications that use the DocumentDB API on my local machine without needing a cloud connection.

By installing the emulator on my Windows machine, I get a local environment that supports document storage, querying, indexing, and authentication. It's a great way to build and test features before deploying them to the actual cloud-based Cosmos DB account.

42. What's the best way to get started with the DocumentDB API?

The best way to get started with the DocumentDB API, which is now known as the SQL API in Cosmos DB, is by following these steps:

1. Install the Azure Cosmos DB emulator if I want to test locally.
2. Use the SDK for my preferred programming language, like .NET, Java, Python, or Node.js.
3. Create a Cosmos DB account in the Azure portal using the SQL API.
4. Use the Azure portal's Data Explorer to create a database and container.
5. Write a simple application that connects to the Cosmos DB account using the connection string and starts inserting or querying data.

Microsoft provides tutorials, code samples, and quickstart guides for each language, which make it easy to begin working with the SQL API. It's also helpful to understand basic JSON document structure and how to use SQL-like queries to work with that data.

43. Is Resource Link Caching Supported by the DocumentDB API?

Yes, resource link caching is supported by the DocumentDB API, or SQL API, in Azure Cosmos DB. When an application interacts with Cosmos DB, it has to access resources like databases, containers, and documents. These resources are identified using URLs or links.

Instead of making a network call every time to look up the location of a resource, the SDK caches these resource links locally. This helps reduce latency and improves the performance of read and write operations, especially when the same resources are accessed repeatedly.

Resource link caching is handled automatically by the SDK, so as a developer, I don't need to do anything extra. It's one of the optimizations that helps Cosmos DB deliver fast and predictable performance.

44. What Are The Table API's Error Messages?

The Table API in Azure Cosmos DB supports common error messages similar to those in Azure Table Storage and general HTTP responses. These error messages help me understand what went wrong during operations. Some of the common error messages are:

- **404 Not Found**: This means the resource I'm trying to access, like a table or an entity, doesn't exist.
- **409 Conflict**: This happens when I try to insert an item that already exists with the same partition key and row key.
- **400 Bad Request**: This is returned when the request is invalid, maybe due to missing fields or wrong data types.
- **403 Forbidden**: This occurs when my authentication is not valid or I don't have permission to access the resource.
- **503 Service Unavailable**: This means the service is temporarily unavailable. I may need to retry after some time.
- **429 Too Many Requests**: This happens when I exceed the provisioned request units. The response includes a retry-after value, telling me how long to wait before trying again.

These error messages help me troubleshoot issues and handle them gracefully in my application by using retries or logging the problem.

45. Describe the differences between relational and NoSQL databases.

Relational databases store data in structured tables with rows and columns, and they use SQL to manage and query data. Each table has a fixed schema, meaning every row must follow the same structure. Data is usually spread across multiple tables and connected using foreign keys and joins.

NoSQL databases, on the other hand, use flexible data models such as documents, key-value pairs, graphs, or wide columns. They do not require a fixed schema, so each record can have a different structure. Instead of joins, data is often stored in a way that makes it easy to retrieve in one read.

Some key differences:

- relational databases use SQL, while NoSQL databases may use different query languages depending on the model
- relational databases are great for structured data and complex transactions
- NoSQL databases are better for large-scale, unstructured, or changing data
- NoSQL databases are designed to scale out easily across many servers

In short, relational databases are good for traditional business applications, while NoSQL databases are often used for modern web, mobile, and real-time apps.

46. With the document data model, why use a NoSQL database?

The document data model stores data in the form of documents, usually in JSON format. This is a perfect match for NoSQL document databases like Azure Cosmos DB because it offers flexibility and scalability.

Here's why I would use a NoSQL database for document data:

- each document can have a different structure, so I don't need to define a fixed schema
- it allows me to store related data together in a single document, which makes it easier and faster to read
- it scales automatically to handle huge volumes of data and high user traffic
- it is easier to change or add new fields without affecting other records
- it supports fast reads and writes, which is helpful for modern applications like e-commerce, IoT, and mobile apps

Overall, NoSQL databases make it simpler to build and maintain applications that need speed, flexibility, and scalability when working with semi-structured or evolving data.

47. What makes Cosmos DB different from other databases?

Cosmos DB is different from other databases because it is a globally distributed, multi-model NoSQL database that is designed for high performance, low latency, and automatic scaling. One of its main strengths is that it can replicate data across multiple Azure regions, which allows applications to run faster and remain available even if a region goes down.

Another difference is that Cosmos DB offers five different APIs, so I can use it like a document database, key-value store, graph database, column-family database, or table store. This flexibility means I can choose the right model for my application without switching platforms.

Cosmos DB also guarantees fast and predictable performance using request units and supports five consistency levels, which gives me control over data accuracy and latency. It automatically indexes all data, so I don't have to manage indexes myself.

In short, Cosmos DB combines global distribution, multi-model support, automatic indexing, multiple consistency levels, and elastic scalability, all in one fully managed service.

48. Explain the different APIs provided by Cosmos DB.

Cosmos DB provides five different APIs, each designed for a specific type of data model or access pattern. These APIs allow me to use Cosmos DB with tools and code that are familiar from other database systems. The APIs are:

1. **SQL API**: This is the default API and is used for working with JSON documents using a SQL-like query language. It's great for document-based applications.
2. **MongoDB API**: This lets me use MongoDB drivers and tools to connect to Cosmos DB. My app can use MongoDB commands and queries without changing much of the code.
3. **Cassandra API**: This allows applications that use Apache Cassandra to run on Cosmos DB. It supports Cassandra query language and data models like keyspace and table.
4. **Gremlin API**: This is for graph-based data, such as social networks or relationship-based queries. It supports the Gremlin query language for working with vertices and edges.
5. **Table API**: This supports key-value data using the same structure as Azure Table Storage. It's simple and good for applications that need fast reads and writes with a flat data structure.

Each API gives me access to the same underlying Cosmos DB features like global distribution, scalability, and low-latency performance, but lets me work in a way that matches the needs of my application.

49. What is the data model used by Cosmos DB and how does it differ from relational databases?

Cosmos DB uses a flexible, schema-less data model. This means data is usually stored in JSON documents, but depending on the API, it can also be stored as key-value pairs, graphs, wide-column records, or tables. The data model adapts to the shape of the data and does not require a fixed structure.

In contrast, relational databases use a strict schema where data is organized into tables with rows and columns. Each row in a table must follow the same structure, and relationships between tables are managed using foreign keys and joins.

The main differences are:

- Cosmos DB allows each document to have a different structure, while relational databases require a fixed format
- Cosmos DB is better for unstructured or semi-structured data, while relational databases are better for structured data
- Cosmos DB avoids joins and instead stores related data in the same document, while relational databases often rely on joining tables
- Cosmos DB is designed to scale horizontally, meaning it can spread data across multiple servers, while relational databases usually scale vertically

This makes Cosmos DB a better fit for applications that need flexibility, speed, and the ability to handle large and changing data sets.

50. Can you describe the multi-model capabilities of Cosmos DB?

Yes, Cosmos DB is called a multi-model database because it supports different ways of storing and accessing data through various APIs. This means I can choose the data model that best fits my application, and still use the same backend service that provides global distribution, high availability, and low latency.

The supported models include:

- **Document model** using the SQL API or MongoDB API, where data is stored as JSON documents. This is good for apps with flexible and dynamic data like user profiles or product catalogs.
- **Key-value model** using the Table API, where data is stored and retrieved using a unique key. This works well for simple lookups.
- **Column-family model** using the Cassandra API, where data is stored in rows with flexible columns. This is useful for time-series data or logs.
- **Graph model** using the Gremlin API, which stores data as vertices and edges. It's ideal for scenarios like social networks or recommendation systems.

Each model is accessed through its own API, but all of them benefit from the same Cosmos DB features like elastic scalability, global distribution, and automatic indexing. This flexibility helps me build different kinds of applications on the same platform.

51. Outline the types of indexing available in Cosmos DB.

Cosmos DB offers automatic indexing by default, which means every property in a document is indexed without needing manual setup. However, it also supports different types of indexing to suit various query needs and data patterns:

- **Range indexing**: This type allows me to perform queries that use greater than, less than, or equal comparisons. It works well with numeric and string fields.
- **Hash indexing**: This type is optimized for equality comparisons. It uses less space and is faster when I only need to match exact values.
- **Spatial indexing**: This is used to support queries on geospatial data, like points, polygons, and distances. It's helpful for location-based apps.
- **Composite indexing**: This allows combining two or more fields in an index to support efficient queries with multiple sort or filter conditions.

I can also create custom indexing policies to include or exclude specific paths, or to control how different data types are indexed. This helps improve performance and reduce storage and write costs when I only need to query specific fields.

52. What is a partition key in Cosmos DB and how is it used?

A partition key in Cosmos DB is a property in each document that determines how the data is distributed across different partitions. It is required for containers that need to scale beyond a single physical partition.

When I create a container, I choose a field in the document as the partition key. For example, if I choose "userId" as the partition key, all documents with the same userId value will be stored in the same logical partition.

The partition key is important for two reasons:

1. **Performance**: Cosmos DB spreads the data and traffic across multiple servers based on the partition key. This helps with fast reads and writes, and avoids bottlenecks.
2. **Scalability**: By using a good partition key with many unique values, I allow Cosmos DB to scale horizontally as the data grows.

A good partition key has high cardinality and distributes both data and traffic evenly. Choosing the right partition key is one of the most important design steps when working with Cosmos DB.

53. Explain the concept of logical partitions vs physical partitions in Cosmos DB.

In Cosmos DB, data is divided into logical partitions and physical partitions to manage scalability and performance efficiently.

A **logical partition** is created based on the value of the partition key. When I choose a partition key for my container, Cosmos DB uses the value of that key to group related documents into the same logical partition. For example, if I use "country" as the partition key, then all documents with the same country value go into the same logical partition.

A **physical partition** is the actual storage unit on the backend infrastructure. Cosmos DB automatically maps multiple logical partitions to a physical partition. Each physical partition can store up to 50 gigabytes of data and handle up to 10,000 request units per second.

As my data grows or as more throughput is needed, Cosmos DB automatically creates new physical partitions and redistributes logical partitions among them. This way, I don't need to manage scaling manually.

54. Enumerate the different consistency levels available in Cosmos DB.

Cosmos DB provides five consistency levels to balance between performance and data accuracy. These levels help decide how up-to-date the data is when it's read after a write operation. The five levels are:

1. **Strong consistency –** The most recent data is always returned. This ensures accuracy but has higher latency and lower availability across regions.
2. **Bounded staleness –** Reads may lag behind writes by a few versions or a specific time window that I define. It balances freshness with better availability than strong consistency.
3. **Session consistency –** Guarantees that within a single session, I will always read my own writes. This is the default level and works well for most applications.
4. **Consistent prefix –** Ensures that data is returned in the exact order it was written. It doesn't guarantee the latest data but always keeps the order correct.
5. **Eventual consistency –** The lowest level. Data might not be updated immediately but will eventually become consistent. This provides the best performance and lowest latency.

55. Explain the trade-offs between consistency, availability, and latency in Cosmos DB.

In Cosmos DB, I need to balance **consistency**, **availability**, and **latency**, which is known as the **CAP theorem**. Here's how the trade-offs work:

- If I choose **strong consistency**, I get the most accurate and up-to-date data, but this might increase latency and reduce availability across regions because all regions must confirm the latest write before returning a read.
- If I choose **eventual consistency**, I get very low latency and high availability, but users might read slightly outdated data right after a write.
- **Session and bounded staleness** offer a middle ground. They provide good accuracy within limits while keeping performance and availability reasonable.

So, if my application is very sensitive to stale data (like a banking system), I might choose strong consistency. But if I care more about speed and can tolerate slight delays (like a product catalog or news feed), I can choose eventual or session consistency.

56. What security features does Cosmos DB provide to protect data?

Cosmos DB offers several security features to protect data at rest and in transit:

1. **Encryption –** All data is encrypted by default using Microsoft-managed keys. I can also choose to use my own customer-managed keys if I need more control.
2. **Authentication and authorization –** Access to Cosmos DB is controlled using master keys, resource tokens, or Azure Active Directory. I can assign roles and permissions to limit what users and apps can do.
3. **Role-based access control (RBAC) –** Using Azure AD, I can define fine-grained access controls for who can read, write, or manage resources in Cosmos DB.
4. **Private endpoints –** I can restrict access to Cosmos DB through private links, so data never travels over the public internet.
5. **Network security –** Cosmos DB supports virtual network service endpoints, IP firewall rules, and built-in denial-of-service protection to prevent unauthorized access.
6. **Compliance –** Cosmos DB complies with major industry standards like HIPAA, GDPR, ISO, and SOC, so it can be used in regulated industries.

These features together help ensure that my data is safe, private, and accessible only by authorized users and applications.