Exam DP-420: Designing and Implementing Cloud-Native Applications Using Microsoft Azure Cosmos DB – Skills Measured

NOTE: Passing score: 700. Learn more about exam scores here.

Audience Profile

A candidate for the Azure Cosmos DB Developer Specialty certification should have subject matter expertise designing, implementing, and monitoring cloud-native applications that store and manage data.

Responsibilities for this role include designing and implementing data models and data distribution, loading data into an Azure Cosmos DB database, and optimizing and maintaining the solution. These professionals integrate the solution with other Azure services. They also design, implement, and monitor solutions that consider security, availability, resilience, and performance requirements.

A candidate for this exam must have solid knowledge and experience developing apps for Azure and working with Azure Cosmos DB database technologies. They should be proficient at developing applications by using the Core (SQL) API and SDKs, writing efficient queries and creating appropriate index policies, provisioning and managing resources in Azure, and creating server-side objects with JavaScript. They should be able to interpret JSON, read C# or Java code, and use PowerShell.

Skills Measured

NOTE: The bullets that follow each of the skills measured are intended to illustrate how we are assessing that skill. This list is NOT definitive or exhaustive.

NOTE: Most questions cover features that are general availability (GA). The exam may contain questions on Preview features if those features are commonly used.

Design and Implement Data Models (35–40%)

Design and implement a non-relational data model for Azure Cosmos DB Core API

- develop a design by storing multiple entity types in the same container
- develop a design by storing multiple related entities in the same document
- develop a model that denormalizes data across documents
- develop a design by referencing between documents
- identify primary and unique keys

- identify data and associated access patterns
- specify a default TTL on a container for a transactional store

Design a data partitioning strategy for Azure Cosmos DB Core API

- choose a partition strategy based on a specific workload
- choose a partition key
- plan for transactions when choosing a partition key
- evaluate the cost of using a cross-partition query
- calculate and evaluate data distribution based on partition key selection
- calculate and evaluate throughput distribution based on partition key selection
- construct and implement a synthetic partition key
- design partitioning for workloads that require multiple partition keys

Plan and implement sizing and scaling for a database created with Azure Cosmos DB

- evaluate the throughput and data storage requirements for a specific workload
- choose between serverless and provisioned models
- choose when to use database-level provisioned throughput
- design for granular scale units and resource governance
- evaluate the cost of the global distribution of data
- configure throughput for Azure Cosmos DB by using the Azure portal

Implement client connectivity options in the Azure Cosmos DB SDK

- choose a connectivity mode (gateway versus direct)
- implement a connectivity mode
- create a connection to a database
- enable offline development by using the Azure Cosmos DB emulator
- handle connection errors
- implement a singleton for the client
- specify a region for global distribution
- configure client-side threading and parallelism options
- enable SDK logging

Implement data access by using the Azure Cosmos DB SQL language

- implement queries that use arrays, nested objects, aggregation, and ordering
- implement a correlated subquery
- implement queries that use array and type-checking functions
- implement queries that use mathematical, string, and date functions
- implement queries based on variable data

Implement data access by using SQL API SDKs

- choose when to use a point operation versus a query operation
- implement a point operation that creates, updates, and deletes documents
- implement an update by using a patch operation
- manage multi-document transactions using SDK Transactional Batch
- perform a multi-document load using SDK Bulk
- implement optimistic concurrency control using ETags
- implement session consistency by using session tokens
- implement a query operation that includes pagination
- implement a query operation by using a continuation token
- handle transient errors and 429s
- specify TTL for a document
- retrieve and use query metrics

Implement server-side programming in Azure Cosmos DB Core API by using JavaScript

- write, deploy, and call a stored procedure
- design stored procedures to work with multiple items transactionally
- implement triggers
- implement a user-defined function

Design and Implement Data Distribution (5–10%)

Design and implement a replication strategy for Azure Cosmos DB

- choose when to distribute data
- define automatic failover policies for regional failure for Azure Cosmos DB Core API
- perform manual failovers to move single master write regions
- choose a consistency model
- identify use cases for different consistency models
- evaluate the impact of consistency model choices on availability and associated RU cost
- evaluate the impact of consistency model choices on performance and latency
- specify application connections to replicated data

Design and implement multi-region write

- choose when to use multi-region write
- implement multi-region write
- implement a custom conflict resolution policy for Azure Cosmos DB Core API

Integrate an Azure Cosmos DB Solution (5–10%)

Enable Azure Cosmos DB analytical workloads

- enable Azure Synapse Link

- choose between Azure Synapse Link and Spark Connector
- enable the analytical store on a container
- enable a connection to an analytical store and query from Azure Synapse Spark or Azure Synapse SQL
- perform a query against the transactional store from Spark
- write data back to the transactional store from Spark

Implement solutions across services

- integrate events with other applications by using Azure Functions and Azure Event Hubs
- denormalize data by using Change Feed and Azure Functions
- enforce referential integrity by using Change Feed and Azure Functions
- aggregate data by using Change Feed and Azure Functions, including reporting
- archive data by using Change Feed and Azure Functions
- implement Azure Cognitive Search for an Azure Cosmos DB solution

Optimize an Azure Cosmos DB Solution (15–20%)

Optimize query performance in Azure Cosmos DB Core API

- adjust indexes on the database
- calculate the cost of the query
- retrieve request unit cost of a point operation or query
- implement Azure Cosmos DB integrated cache

Design and implement change feeds for an Azure Cosmos DB Core API

- develop an Azure Functions trigger to process a change feed
- consume a change feed from within an application by using the SDK
- manage the number of change feed instances by using the change feed estimator
- implement denormalization by using a change feed
- implement referential enforcement by using a change feed
- implement aggregation persistence by using a change feed
- implement data archiving by using a change feed

Define and implement an indexing strategy for an Azure Cosmos DB Core API

- choose when to use a read-heavy versus write-heavy index strategy
- choose an appropriate index type
- configure a custom indexing policy by using the Azure portal
- implement a composite index
- optimize index performance

Maintain an Azure Cosmos DB Solution (25–30%)

Monitor and troubleshoot an Azure Cosmos DB solution

- evaluate response status code and failure metrics
- monitor metrics for normalized throughput usage by using Azure Monitor
- monitor server-side latency metrics by using Azure Monitor
- monitor data replication in relation to latency and availability
- configure Azure Monitor alerts for Azure Cosmos DB
- implement and query Azure Cosmos DB logs
- monitor throughput across partitions
- monitor distribution of data across partitions
- monitor security by using logging and auditing

Implement backup and restore for an Azure Cosmos DB solution

- choose between periodic and continuous backup
- configure periodic backup
- configure continuous backup and recovery
- locate a recovery point for a point-in-time recovery
- recover a database or container from a recovery point

Implement security for an Azure Cosmos DB solution

- choose between service-managed and customer-managed encryption keys
- configure network-level access control for Azure Cosmos DB
- configure data encryption for Azure Cosmos DB
- manage control plane access to Azure Cosmos DB by using Azure role-based access control (RBAC)
- manage data plane access to Azure Cosmos DB by using keys
- manage data plane access to Azure Cosmos DB by using Azure Active Directory
- configure Cross-Origin Resource Sharing (CORS) settings
- manage account keys by using Azure Key Vault
- implement customer-managed keys for encryption
- implement Always Encrypted

Implement data movement for an Azure Cosmos DB solution

- choose a data movement strategy
- move data by using client SDK bulk operations
- move data by using Azure Data Factory and Azure Synapse pipelines
- move data by using a Kafka connector
- move data by using Azure Stream Analytics
- move data by using the Azure Cosmos DB Spark Connector

Implement a DevOps process for an Azure Cosmos DB solution

- choose when to use declarative versus imperative operations
- provision and manage Azure Cosmos DB resources by using Azure Resource Manager templates (ARM templates)
- migrate between standard and autoscale throughput by using PowerShell or Azure CLI
- initiate a regional failover by using PowerShell or Azure CLI
- maintain index policies in production by using ARM templates