DUMPSARENA

# Designing and Implementing Cloud-Native Applications Using Microsoft Azure Cosmos DB

**Microsoft DP-420**

**Version Demo**

**Total Demo Questions: 23**

**Total Premium Questions: 233**

**Buy Premium PDF**

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

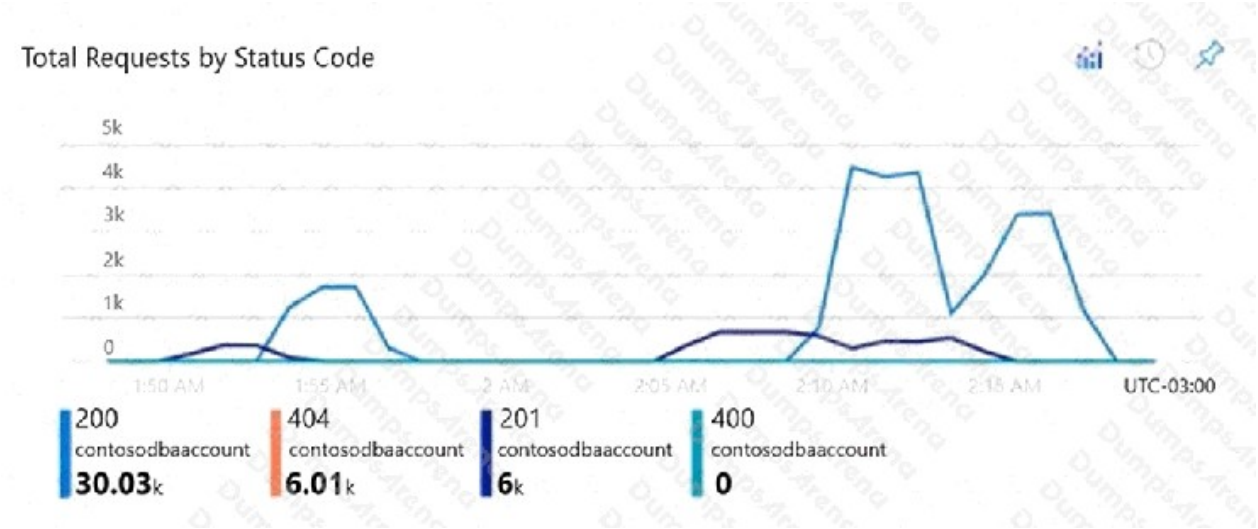
# Topic Break Down

| Topic | No. of Questions |
|---|---|
| Topic 1, Design and implement data models | 49 |
| Topic 2, Design and implement data distribution | 39 |
| Topic 3, Integrate an Azure Cosmos DB solution | 70 |
| Topic 4, Optimize an Azure Cosmos DB solution | 27 |
| Topic 5, Maintain an Azure Cosmos DB solution | 41 |
| Topic 6, Case Study 1 | 4 |
| Topic 7, Case Study 2 | 3 |
| Total | 233 |

QUESTION NO: 1 - (SIMULATION)

You have an Azure Cosmos DB Core (SQL) API account used by an application named App1.

You open the Insights pane for the account and see the following chart.




Use the drop-down menus to select the answer choice that answers each question based on the information presented in the graphic.

NOTE: Each correct selection is worth one point.

The HTTP 404 status code is caused by **[answer choice]**

- incorrect connection URLs
- an intermittent firewall issue
- incorrectly formatted partition keys
- requesting resources that do not exist

There are **[answer choice]** successful resource creations in the account during the time period of the chart

- zero
- 6 thousand
- 6.01 thousand
- 30.03 thousand
- 36.03 thousand

**ANSWER: See the explanation for the answer**

**Explanation:**

**Select:**

A `404 Not Found` response means the requested Cosmos DB resource cannot be found. A `201 Created` response denotes successful resource creation, and the chart shows `201` requests totaling **6k**.

## QUESTION NO: 2

You have an application named App1 that reads the data in an Azure Cosmos DB Core (SQL) API account. App1 runs the same read queries every minute. The default consistency level for the account is set to eventual.

You discover that every query consumes request units (RUs) instead of using the cache.

You verify the IntegratedCacheiteItemHitRate metric and the IntegratedCacheQueryHitRate metric. Both metrics have values of 0.

You verify that the dedicated gateway cluster is provisioned and used in the connection string.

You need to ensure that App1 uses the Azure Cosmos DB integrated cache.

What should you configure?

**A.** the indexing policy of the Azure Cosmos DB container

**B.** the consistency level of the requests from App1

**C.** the connectivity mode of the App1 CosmosClient

**D.** the default consistency level of the Azure Cosmos DB account

**ANSWER: C**

**Explanation:**

The correct configuration is **the connectivity mode of the App1 CosmosClient**. Azure Cosmos DB integrated cache is hosted in the dedicated gateway cluster, but merely provisioning that cluster and using its endpoint is not sufficient if the SDK client is configured for Direct mode. App1 must create its CosmosClient with Gateway connection mode so that read requests are routed through the dedicated gateway and can be evaluated against the integrated cache. When a repeated point read or query is served from this cache, Azure Cosmos DB avoids charging RUs for the backend read operation, aside from any applicable gateway processing behavior.

The account's eventual consistency default supports integrated-cache use because the cache is available for eventual and session-consistent reads. Once Gateway mode is configured, repeated query requests can produce integrated-cache hits while cached results remain within the configured maximum staleness interval. The IntegratedCacheItemHitRate and IntegratedCacheQueryHitRate metrics can then be used to confirm that the application is receiving cache hits. See Azure Cosmos DB integrated cache overview and configure the integrated cache.

## QUESTION NO: 3 - (HOTSPOT)

You have an Azure Cosmos DB container named container! that has a provisioned throughput and two physical partitions. You monitor the following metrics for container1

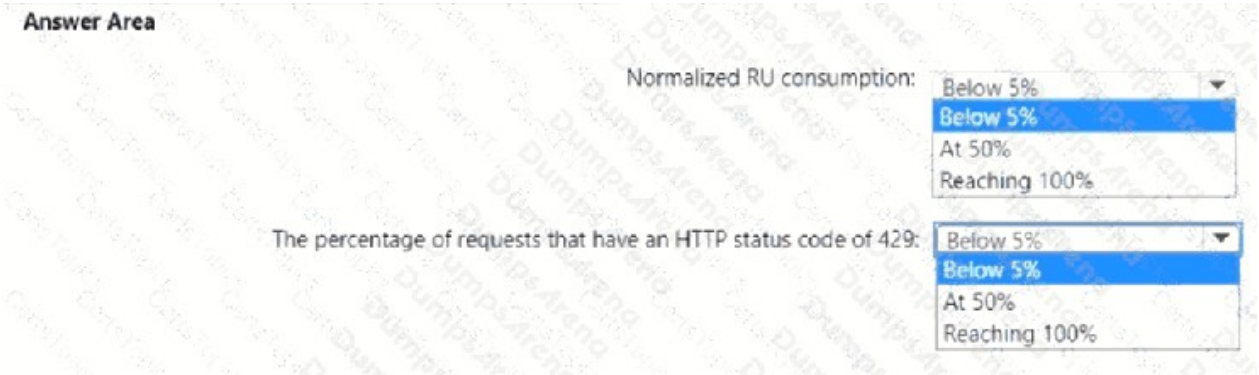
• Normalized RU consumption

• The percentage of requests that have an HTTP status code of 429

You need to confirm that container1 is configured to maximize resource utilization.

What are the optimal values for each metric? To answer, select the appropriate options in the answer area.

NOTE: Each correct selection is worth one point.




**ANSWER:**




**Explanation:**

To maximize utilization of provisioned throughput, select **Reaching 100%** for Normalized RU consumption and **Below 5%** for the percentage of requests that have an HTTP status code of 429.

Normalized RU consumption expresses the RU/s used by the busiest physical partition as a percentage of the RU/s available to that partition. Since provisioned throughput is distributed among physical partitions, this metric is especially useful for determining whether any partition has reached its available capacity. A value reaching 100% confirms that the busiest partition is consuming all of its assigned throughput, which is the target when the goal is to maximize resource use. Microsoft describes normalized RU consumption as a way to identify how much throughput is being consumed relative to the provisioned amount, including at the physical-partition level.

A small rate of throttled requests is also expected at efficient maximum utilization. Azure Cosmos DB returns HTTP 429 when a request exceeds the available RU/s at that moment, and clients are designed to retry these requests after the service-provided retry interval. Microsoft guidance identifies an overall 429 rate below 5% as a healthy target; it indicates that the application is using the provisioned capacity effectively while throttling remains limited. Together, a 100% normalized peak and fewer than 5% HTTP 429 responses show that the container is operating at high utilization without excessive throttling.

For additional details, see Monitor normalized request unit consumption in Azure Cosmos DB and Diagnose and troubleshoot Azure Cosmos DB HTTP 429 errors.

## QUESTION NO: 4

You have an Azure Cosmos DB for NoSQL account that uses session consistency.

An application writes an item and then reads the item from a different region.

You need to ensure that the application can read its own successful writes.

What should the application use?

**A.** The session token returned by the write operation

**B.** The continuation token returned by the write operation

**C.** The entity tag of the written item

**D.** The resource token for the container

## ANSWER: A

**Explanation:**

The application should use the session token returned by the write operation when it performs subsequent reads. Session consistency provides read-your-writes guarantees for a client session when the client preserves and sends the session token. The token identifies the logical sequence of writes that must be visible to the read request.

This is especially important when a subsequent read can be routed to another region. By including the session token, the client ensures that Azure Cosmos DB does not return a version older than the write acknowledged within that session. The

Azure Cosmos DB SDK manages session tokens automatically when the same client instance is used, but applications that transfer work across client instances or service boundaries can propagate the token explicitly.

See Consistency levels in Azure Cosmos DB.

## QUESTION NO: 5

Note: This question is part of a series of questions that present the same scenario. Each question in the series contains a unique solution that might meet the stated goals. Some question sets might have more than one correct solution, while others might not have a correct solution.

After you answer a question in this section, you will NOT be able to return to it. As a result, these questions will not appear in the review screen.

You have an Azure Cosmos DB Core (SQL) API account named account 1 that uses autoscale throughput.

You need to run an Azure function when the normalized request units per second for a container in account1 exceeds a specific value.

Solution: You configure an Azure Monitor alert to trigger the function.

Does this meet the goal?

**A.** Yes

**B.** No

## ANSWER: A

**Explanation:**

Yes. Azure Cosmos DB publishes the *Normalized RU Consumption* metric to Azure Monitor. This metric represents the percentage of provisioned throughput consumed during an interval. For an account using autoscale throughput, the percentage is calculated relative to the maximum autoscale RU/s configured for the resource, making it suitable for detecting when a container is approaching a defined throughput-consumption level.

An Azure Monitor metric alert rule can evaluate this metric for the relevant Cosmos DB resource and apply a threshold condition. The alert rule can then invoke an action group when the threshold is breached. Azure Monitor action groups support an Azure Function action, allowing the alert to call an Azure Function automatically. Therefore, configuring a metric alert for normalized RU consumption and associating it with an action group that invokes the function fulfills the requirement to run the function when consumption exceeds the specified value.

For Cosmos DB metric definitions and normalized RU consumption behavior, see Azure Cosmos DB monitoring data reference. For Azure Function actions in alert action groups, see Azure Monitor action groups.

## QUESTION NO: 6

You have the following query.

```
SELECT * FROM c

WHERE c.sensor = "TEMP1"

AND c.value < 22

AND c.timestamp >= 1619146031231
```

You need to recommend a composite index strategy that will minimize the request units (RUs) consumed by the query.

What should you recommend?

**A.** a composite index for (sensor ASC, value ASC) and a composite index for (sensor ASC, timestamp ASC)

**B.** a composite index for (sensor ASC, value ASC, timestamp ASC) and a composite index for (sensor DESC, value DESC, timestamp DESC)

**C.** a composite index for (value ASC, sensor ASC) and a composite index for (timestamp ASC, sensor ASC)

**D.** a composite index for (sensor ASC, value ASC, timestamp ASC)

**ANSWER: A**

**Explanation:**

**a composite index for (sensor ASC, value ASC) and a composite index for (sensor ASC, timestamp ASC)** is the appropriate strategy because the query applies an equality filter to `sensor` and range filters to both `value` and `timestamp`. Azure Cosmos DB composite indexes can accelerate queries containing equality predicates followed by a range predicate. For a query with more than one range-filtered property, Cosmos DB can use multiple composite indexes: one index should pair the equality-filtered property with each range-filtered property. Placing `sensor` first lets the index narrow the candidate items to the requested sensor value, while the respective second paths support the `value < 22` and `timestamp >= 1619146031231` predicates. This reduces the amount of index and document data that must be scanned and therefore can reduce RU consumption. Ascending order is suitable here because Cosmos DB can use a composite index in either sort direction when the query does not include an `ORDER BY` clause. Microsoft documents this multiple-range-filter pattern and recommends defining a composite index for each relevant range property after the equality-filtered paths. See Microsoft Learn: composite indexes and Microsoft Learn: composite indexing policy examples.

**QUESTION NO: 7**

You are developing an application that will use an Azure Cosmos DB Core (SQL) API account as a data source.

You need to create a report that displays the top five most ordered fruits as shown in the following table.

| Name | Type | Orders |
|---|---|---|
| apple | fruit | 1,000 |
| orange | fruit | 600 |
| banana | fruit, exotic | 400 |
| plum | fruit | 300 |
| mango | fruit, exotic | 200 |

A collection that contains aggregated data already exists. The following is a sample document:

{

"name": "apple",

"type": ["fruit", "exotic"],

"orders": 10000 }

Which two queries can you use to retrieve data for the report? Each correct answer presents a complete solution.

NOTE: Each correct selection is worth one point.

A.
```
SELECT TOP  i.name, i.types, i.orders
FROM items i
WHERE EXISTS(SELECT VALUE t FROM t IN i.types WHERE t.name = 'fruit')
ORDER BY i.orders,i.types
```

B.
```
SELECT TOP  i.name, i.types, i.orders
FROM items i
WHERE EXISTS(SELECT VALUE t FROM t IN i.types WHERE t.name = 'fruit')
ORDER BY i.orders DESC
```

C.
```
SELECT TOP  i.name, i.types, i.orders
FROM items i
WHERE EXISTS(SELECT VALUE t FROM t IN i.types WHERE t.name = 'fruit')
ORDER BY i.types DESC
```

D.
```
SELECT TOP  i.name, i.types, i.orders
FROM items i
WHERE ARRAY_CONTAINS(i.types, {name: 'fruit'})
ORDER BY i.orders DESC
```

---

**A.** SELECT TOP 5 c.name, c.orders FROM c WHERE ARRAY_CONTAINS(c.type, "fruit") ORDER BY c.orders

**B.** SELECT TOP 5 c.name, c.orders FROM c WHERE ARRAY_CONTAINS(c.type, "fruit") ORDER BY c.orders DESC

**C.** SELECT TOP 5 c.name, c.orders FROM c WHERE ARRAY_CONTAINS(c.type, "fruit") ORDER BY c.type DESC

**D.** SELECT TOP 5 c.name, c.orders FROM c JOIN t IN c.type WHERE t = "fruit" ORDER BY c.orders DESC

**ANSWER: B D**

**Explanation:**

The query `SELECT TOP 5 c.name, c.orders FROM c WHERE ARRAY_CONTAINS(c.type, "fruit") ORDER BY c.orders DESC` produces the required report because `ARRAY_CONTAINS` tests whether the document's `type` array contains the scalar value `"fruit"`. The filter therefore includes documents such as the sample apple document, even when fruit is only one of several classifications. Sorting by `c.orders DESC` places the largest order totals first, and `TOP 5` limits the returned result set to the five highest-ranking fruits.

The query using `JOIN t IN c.type` is also valid. This JOIN iterates the elements of each document's `type` array, allowing `WHERE t = "fruit"` to select fruit-classified documents. It then applies the same descending order on the numeric `orders` property and returns only the first five rows. Both approaches use supported Azure Cosmos DB for NoSQL query syntax for filtering array values and ordering result sets. See ARRAY_CONTAINS and ORDER BY in Azure Cosmos DB for NoSQL.

## QUESTION NO: 8

You have an Azure Cosmos DB for NoSQL account that uses provisioned throughput.

You need to create Azure Monitor alerts that identify when the workload is approaching the available throughput limit and when requests are being rate limited.

Which two metrics should you use? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Normalized RU Consumption

**B.** Total Requests filtered by status code 429

**C.** Data Usage

**D.** Document Quota

**E.** Availability

**ANSWER: A B**

**Explanation:**

Use **Normalized RU Consumption** to identify throughput pressure. This metric represents the percentage of provisioned throughput consumed by the busiest physical partition. Values approaching 100 percent can indicate that requests are at risk of rate limiting.

Use **Total Requests** and filter or split the metric by the 429 status-code dimension to identify requests that were rate limited. A 429 response indicates that the request exceeded the throughput available at that time.

Data Usage measures consumed storage capacity rather than request-unit utilization. Document Quota relates to storage limits, and Availability is not a measure of rate limiting or throughput saturation.

See Azure Cosmos DB monitoring data reference and Create alerts for Azure Cosmos DB.

**QUESTION NO: 9**

You have an application that writes items to an Azure Cosmos DB for NoSQL container.

During periods of high demand, the application receives HTTP status code 429 responses.

You need to minimize failed write operations without increasing the provisioned throughput.

Which two actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Configure the application to retry requests that receive HTTP status code 429.

**B.** Honor the retry-after interval returned in the response.

**C.** Set the maximum retry attempts to zero.

**D.** Retry each request immediately after receiving a 429 response.

**E.** Change the consistency level of write requests to eventual.

**ANSWER: A B**

**Explanation:**

Configure the application to retry requests that receive HTTP status code 429 and honor the retry-after interval returned by Azure Cosmos DB. A 429 response indicates that the request was rate limited because the available request-unit throughput was consumed. The response includes a retry-after value that identifies when the client should retry the request.

Use an Azure Cosmos DB SDK retry policy. Current Azure Cosmos DB SDKs automatically retry rate-limited requests by default. Configuring the retry policy appropriately allows the application to tolerate transient periods of throughput pressure while avoiding immediate repeated requests that are likely to be rate limited again.

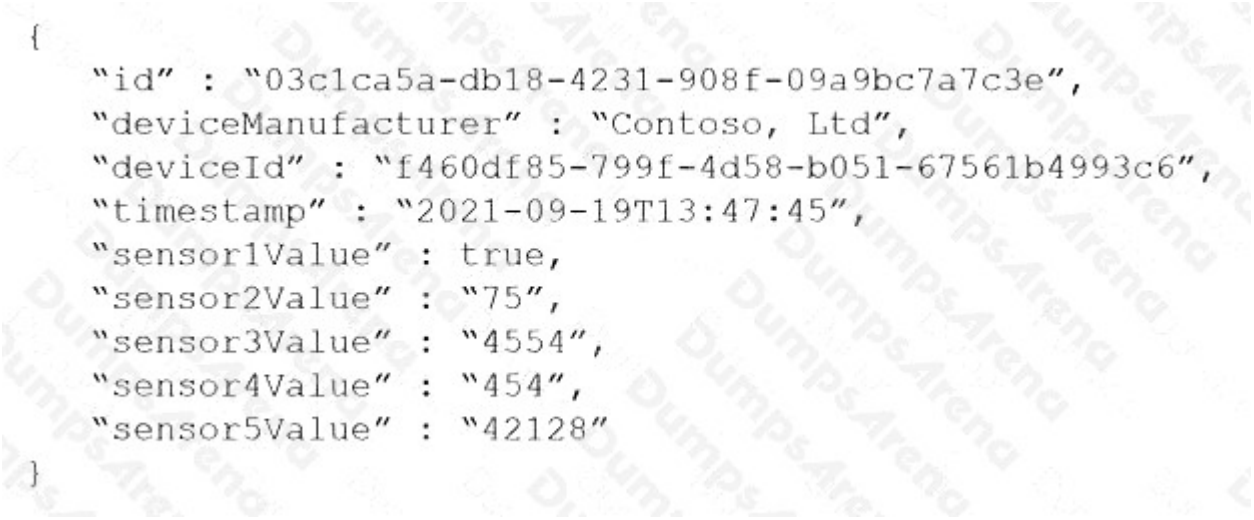
Reducing the number of retries to zero does not resolve throttling. Retrying immediately without observing the service-provided delay can increase contention. Changing the consistency level does not address rate limiting of write requests.

See Troubleshoot request rate too large or HTTP 429 errors in Azure Cosmos DB for NoSQL.

## QUESTION NO: 10

You are designing an Azure Cosmos DB Core (SQL) API solution to store data from IoT devices. Writes from the devices will be occur every second.

The following is a sample of the data.



```
{
    "id" : "03c1ca5a-db18-4231-908f-09a9bc7a7c3e",
    "deviceManufacturer" : "Contoso, Ltd",
    "deviceId" : "f460df85-799f-4d58-b051-67561b4993c6",
    "timestamp" : "2021-09-19T13:47:45",
    "sensor1Value" : true,
    "sensor2Value" : "75",
    "sensor3Value" : "4554",
    "sensor4Value" : "454",
    "sensor5Value" : "42128"
}
```

You need to select a partition key that meets the following requirements for writes:

Minimizes the partition skew

Avoids capacity limits

Avoids hot partitions

What should you do?

**A.** Use timestamp as the partition key.

**B.** Create a new synthetic key that contains deviceId and sensor1Value.

**C.** Create a new synthetic key that contains deviceId and deviceManufacturer.

**D.** Create a new synthetic key that contains deviceId and a random number.

## ANSWER: D

**Explanation:**

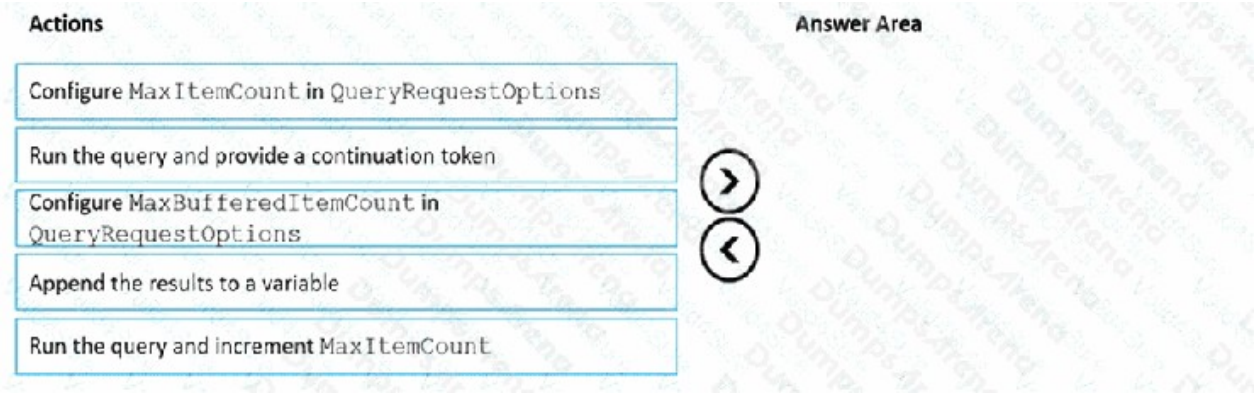
Create a new synthetic key that contains deviceId and a random number. This design deliberately increases the cardinality and distribution of partition-key values for high-frequency IoT writes. Rather than directing every record associated with one device to one logical partition, appending a random value spreads that device's writes across multiple logical and physical partitions. This enables parallel write processing, reducing the likelihood that a disproportionately active device, or a small set of active devices, consumes the throughput available to a single partition. It therefore helps avoid hot partitions and partition skew while allowing the workload to scale beyond per-partition throughput and storage boundaries. In Azure Cosmos DB, a partition key should distribute request units and storage evenly; synthetic partition keys are an established technique when a naturally occurring property does not provide sufficient distribution. The random component should be generated as part of the stored synthetic key, and applications should account for querying multiple values when retrieving all records for a device. See Azure Cosmos DB partitioning overview and synthetic partition keys in Azure Cosmos DB for NoSQL.

## QUESTION NO: 11 - (DRAG DROP)

You have an app that stores data in an Azure Cosmos DB Core (SQL) API account The app performs queries that return large result sets.

You need to return a complete result set to the app by using pagination. Each page of results must return 80 items.

Which three actions should you perform in sequence? To answer, move the appropriate actions from the list of actions to the answer area and arrange them in the correct order.




**ANSWER:**

**Explanation:**

Set `MaxItemCount` in `QueryRequestOptions` first, using a value of 80. This setting supplies the requested upper limit for the number of documents returned in an individual response page. Azure Cosmos DB can return fewer documents when required by response-size, execution-time, or available-throughput constraints, but this is the setting that requests pages of up to 80 items. The request options are established before the query iterator is created or read, so they apply consistently to pagination.

Next, run the query while using the continuation-token workflow. Azure Cosmos DB query execution is stateless on the service, and a continuation token represents the query position after the current response. The application reads a page, obtains the token when more data remains, and supplies that token on the next request to continue from precisely the next unread result. This is the supported approach for retrieving a large SQL API query result over multiple pages without attempting to return the entire result set in one response. Microsoft documents this behavior in Pagination in Azure Cosmos DB for NoSQL and documents the request option in the QueryRequestOptions.MaxItemCount API reference.

Finally, append the resources returned by every page to a variable or collection in the application. Repeating the continuation-token reads and appending each page preserves all results across the paginated operation. Once no continuation token is returned, the collection contains the complete query result set, assembled from pages requested at the intended size of 80 items.

## QUESTION NO: 12

You have an Azure Cosmos DB for NoSQL account that has multiple write regions.

You need to receive an alert when requests that target the database exceed the available request units per second (RU/s).

Which Azure Monitor signal should you use?

**A.** Region Removed

**B.** Provisioned Throughput

**C.** Metadata Requests

**D.** Data Usage

**E.** Normalized RU Consumption

**ANSWER: E**

**Explanation:**

**Normalized RU Consumption** is the appropriate Azure Monitor metric for this alert. It represents the RU consumption as a percentage of the provisioned throughput, with the reported value reflecting the highest normalized RU use across physical partitions. This makes it the key signal for identifying when requests are approaching or exceeding the RU/s capacity available to the workload. An alert can be configured against this metric using a threshold appropriate to the operational objective, such as a sustained value near 100 percent.

For an account with multiple write regions, evaluate the metric by region as needed. Azure Cosmos DB provisioned throughput and request processing are region-specific, so monitoring normalized consumption with the relevant regional dimension helps identify capacity pressure in a particular write region. The metric is particularly useful because throttling can occur when a physical partition reaches its available RU/s, even when aggregate account-level usage appears lower. Microsoft documents *Normalized RU Consumption* as the metric to monitor provisioned-throughput utilization and to help detect situations that can lead to rate limiting. See Azure Cosmos DB monitoring data reference and Troubleshoot request rate too large exceptions.

## QUESTION NO: 13

You have a database named db1 in an Azure Cosmos DB f You have a third-party application that is exposed thro You need to migrate data from the application to a What should you use?

**A.** Database Migration Assistant

**B.** Azure Data Factory

**C.** Azure Migrate

## ANSWER: B

**Explanation:**

Azure Data Factory is the appropriate service for migrating data exposed by a third-party application through an OData endpoint into Azure Cosmos DB for NoSQL. Azure Data Factory provides a managed, scalable data-integration platform and includes an OData connector that can be used as the source in a Copy activity. The activity can retrieve entities from the application's OData feed, apply supported mapping as needed, and write the resulting documents to an Azure Cosmos DB destination.

This approach is suitable for a one-time migration as well as repeatable or scheduled ingestion. A pipeline can define the source connection, Cosmos DB connection, destination container, column-to-property mappings, and operational settings such as batch behavior. Azure Data Factory also supplies monitoring and execution history, which helps validate that the migration has completed successfully and identify any records that require follow-up. Microsoft documents OData as a supported Azure Data Factory connector and documents Azure Cosmos DB for NoSQL as a supported Copy activity destination. See the Azure Data Factory OData connector documentation and the Azure Cosmos DB for NoSQL connector documentation.

## QUESTION NO: 14

You have an Azure Cosmos DB for NoSQL container named sessions. The container stores session items that should expire automatically after 24 hours.

You need to configure time to live (TTL) for the container and ensure that individual session items use the 24-hour expiration period.

Which two actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Set the container default TTL to `86400`.

**B.** Set the `ttl` property of each session item to `86400`.

**C.** Set the container default TTL to `-1`.

**D.** Set the `ttl` property of each session item to `-1`.

**E.** Set the `ttl` property of each session item to `0`.

**ANSWER: A B**

**Explanation:**

Enable TTL on the container by setting its default TTL to 86,400 seconds. A positive default TTL value enables expiration and specifies the expiration interval used by items that do not define their own `ttl` property.

To ensure that an individual item explicitly uses the same 24-hour interval, set the item's `ttl` property to 86,400. When an item-level TTL value is present, it overrides the container default for that item. Azure Cosmos DB calculates expiration from the item's last modified time.

A value of `-1` indicates that an item does not expire. A value of `0` does not configure a 24-hour retention interval.

See Time to live in Azure Cosmos DB for NoSQL.

**QUESTION NO: 15**

You have a container m an Azure Cosmos DB for NoSQL account.

Data update volumes are unpredictable.

You need to process the change teed of the container by using a web app that has multiple instances. The change feed will be processed by using the change feed processor from the Azure Cosmos DB SDK. **The** multiple instances must share the workload.

Which three actions should you perform? Each correct answer presents part of the solution.

**NOTE:** Each correct selection is worth one point.

**A.** Configure the same processor name for all the instances.

**B.** Configure a different processor name for each instance.

**C.** Configure a different lease container configuration for each instance.

**D.** Configure the same instance name for all the instances. 13

**E.** Configure a different instance name for each instance.

**F.** Configure the same lease container configuration for all the instances.

**ANSWER: A E F**

**Explanation:**

Configure the same processor name for all the instances, configure a different instance name for each instance, and configure the same lease container configuration for all the instances. Together, these settings let all web-app instances participate in one coordinated change feed processor deployment. The processor name identifies the logical processor and is used when managing its lease state. Using the same name ensures that every worker is operating against the same processing group rather than creating independent groups that could each process the feed separately.

The lease container is the shared coordination store for that processor group. Every instance must point to the same lease container configuration so that it can read and update the same lease documents. Those leases represent ownership of monitored feed ranges and enable the SDK to distribute ranges among active workers. Each running worker must also have its own instance name. This gives the change feed processor a distinct host identity for lease ownership, renewals, and rebalancing. As instances are added or removed in response to unpredictable update volume, the processor automatically balances lease ownership across the active instances.

For implementation guidance, see Change feed processor in Azure Cosmos DB for NoSQL and Change feed processor components.

## QUESTION NO: 16

You have an Azure Cosmos DB Core (SQL) API account that is used by 10 web apps.

You need to analyze the data stored in the account by using Apache Spark to create machine learning models. The solution must NOT affect the performance of the web apps.

Which two actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** In an Apache Spark pool in Azure Synapse, create a table that uses cosmos.olap as the data source.

**B.** Create a private endpoint connection to the account.

**C.** In an Azure Synapse Analytics serverless SQL pool, create a view that uses OPENROWSET and the CosmosDB provider.

**D.** Enable Azure Synapse Link for the account and Analytical store on the container.

**E.** In an Apache Spark pool in Azure Synapse, create a table that uses cosmos.oltp as the data source.

## ANSWER: A D

**Explanation:**

Enable Azure Synapse Link for the account and Analytical store on the container. Azure Synapse Link creates and maintains an analytical store alongside the transactional store in Azure Cosmos DB. The analytical store is column-oriented and is automatically synchronized from operational data without requiring an ETL pipeline. It provides an isolated data representation specifically intended for large-scale analytics, so analytical reads do not consume request units or contend with the transactional workload serving the web apps.

In an Apache Spark pool in Azure Synapse, create a table that uses cosmos.olap as the data source. The `cosmos.olap` Spark connector source reads from the Azure Cosmos DB analytical store enabled through Synapse Link. This lets Spark access the container's analytical data for transformations, model training, and other machine-learning workloads while preserving isolation from the operational store. Together, these actions establish an HTAP architecture: web applications continue to use the transactional store, while Apache Spark performs analytics against the separate analytical store. See Azure Synapse Link for Azure Cosmos DB and query the analytical store with Apache Spark.

## QUESTION NO: 17

You use the Azure Cosmos DB change feed processor to process changes from a container named orders.

You need to deploy three instances of the processor. The solution must distribute partitions among the instances and ensure that the instances act as a single processing group.

Which three actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Configure each instance to use the same lease container.

**B.** Configure each instance to use the same lease container prefix.

**C.** Configure each instance with a unique instance name.

**D.** Configure each instance to use a different lease container.

**E.** Configure each instance to use a different lease container prefix.

**ANSWER: A B C**

**Explanation:**

All instances in the same change feed processor group must use the same lease container and the same lease container prefix. Lease documents store partition ownership and continuation state. When the instances use the same lease configuration, the change feed processor coordinates ownership so that a lease is processed by only one active instance at a time.

Each processor instance must also have a unique instance name. The processor uses this identity when acquiring and renewing leases. If an instance stops or becomes unavailable, another instance can acquire its leases and continue processing from the stored continuation state.

Using different lease prefixes creates independent processor groups. That configuration is appropriate when separate applications must each receive all changes, but it does not distribute one shared processing workload among the instances.

See Change feed processor in Azure Cosmos DB for NoSQL.

## QUESTION NO: 18

You are building an application that will store data in an Azure Cosmos DB for NoSQL account. The account uses the session default consistency level. The account is used by five other applications. The account has a single read-write region and 10 additional read regions.

Approximately 20 percent of the items stored in the account are updated hourly.

Several users will access the new application from multiple devices.

You need to ensure that the users see the same item values consistently when they browse from the different devices. The solution must not affect the other applications.

Which two actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Use implicit session management when performing read requests.

**B.** Provide a stored session token when performing read requests.

**C.** Associate a session token to the user account.

**D.** Set the default consistency level to eventual.

**E.** Associate a session token to the device.

**ANSWER: B C**

**Explanation:**

Session consistency provides read-your-writes and monotonic-read guarantees within a logical client session. Normally, Azure Cosmos DB SDKs manage session tokens implicitly in memory for a single client instance. That approach does not carry a user's session state to another device or to a newly created application client. To maintain a consistent experience as a user moves between devices, the application should persist the latest session token as part of that user's application state. Therefore, associating a session token to the user account enables the new device to retrieve the user's prior session context. The application must then provide that stored session token with subsequent read requests, allowing Azure Cosmos DB to serve a version that is at least as recent as the version already observed by that user. This solution uses session-level behavior within the new application rather than changing the account-level default consistency configuration, so the five existing applications remain unaffected. Session tokens are scoped to the relevant container/feed range and should be updated whenever a newer token is returned by a read or write operation. Microsoft documents both the guarantees of session consistency and the use of session tokens for externally managed sessions in Azure Cosmos DB consistency levels and Manage consistency in Azure Cosmos DB for NoSQL.

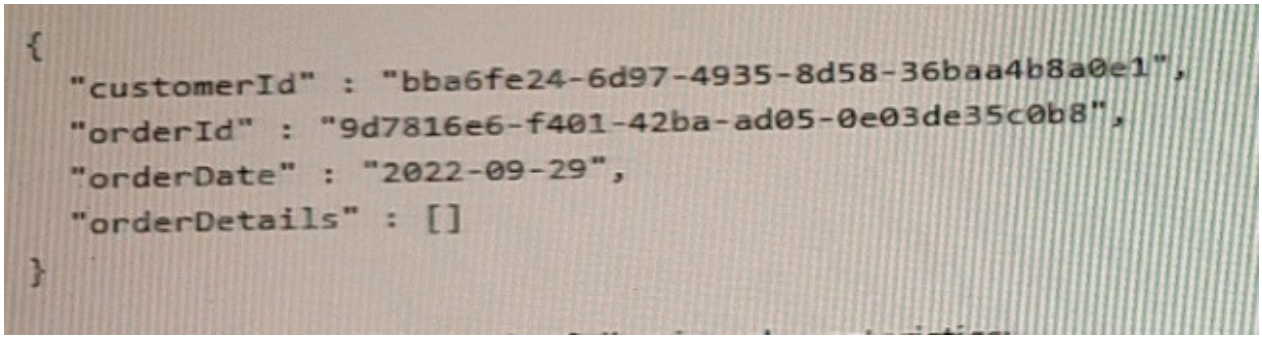
## QUESTION NO: 19

You have a database named db1 in an Azure Cosmos DB for NoSQL

You are designing an application that will use dbl.

In db1, you are creating a new container named coll1 that will store in coll1.

The following is a sample of a document that will be stored in coll1.




The application will have the following characteristics:

• New orders will be created frequently by different customers.

• Customers will often view their past order history.

You need to select the partition key value for coll1 to support the application. The solution must minimize costs.

To what should you set the partition key?

**A.** id

**B.** customerId

**C.** orderDate

**D.** orderId

## ANSWER: B

**Explanation:**

`customerId` is the appropriate partition key because it aligns the container's data distribution with the application's primary read pattern: retrieving a customer's previous orders. When each order includes the customer's identifier as its partition-key value, all orders for an individual customer are logically grouped in the same partition-key scope. The application can then issue targeted queries for that customer's order history by supplying the partition-key value, avoiding fan-out queries across multiple logical partitions. This reduces request-unit consumption and therefore helps minimize cost.

It also supports the write pattern well when orders are created by many different customers. A customer identifier typically has many distinct values, allowing Azure Cosmos DB to distribute order items across logical and physical partitions as the container grows. The partition key should be selected based on both workload distribution and the most frequent query patterns; using `customerId` provides efficient point reads or partition-scoped queries for the stated customer-history scenario. Microsoft recommends choosing a partition key with high cardinality that evenly distributes request units and storage, while enabling efficient queries that include the partition key. See Partitioning and horizontal scaling in Azure Cosmos DB and Query items in Azure Cosmos DB for NoSQL.

## QUESTION NO: 20

You have an Azure Cosmos DB for NoSQL account that supports an operational application.

You need to use Azure Synapse Analytics to run Apache Spark analytics against container data without consuming request units from the transactional workload.

Which two actions should you perform? Each correct answer presents part of the solution.

NOTE: Each correct selection is worth one point.

**A.** Enable Azure Synapse Link and analytical store for the container.

**B.** Use cosmos.olap as the Apache Spark data source.

**C.** Use cosmos.oltp as the Apache Spark data source.

**D.** Provision a dedicated gateway and integrated cache.

**E.** Disable indexing on the transactional container.

**ANSWER: A B**

**Explanation:**

Enable Azure Synapse Link and enable the analytical store on the required container. Azure Synapse Link maintains a column-oriented analytical store that is synchronized from the transactional store. Analytical queries against that store do not consume request units from the transactional workload.

Use the `cosmos.olap` data source from an Apache Spark pool to access the analytical store. The `cosmos.olap` connector is designed for analytical-store access. In contrast, `cosmos.oltp` accesses the transactional store and consumes transactional RUs.

Creating a dedicated gateway and integrated cache can reduce costs for repeated operational reads, but it does not provide the analytical-store isolation required for Spark analytics.

See Azure Synapse Link for Azure Cosmos DB and Query the analytical store with Apache Spark.

**QUESTION NO: 21**

You have an Azure Cosmos DB for NoSQL account. The account hosts a container that has the change feed enabled. You are building an app by using the Azure Cosmos DB SDK. The app will read items from the change feed by using a pull model. You need to ensure that multiple threads can read the change feed in parallel. What should you include?

**A.** the changeFeedStartFew value

**B.** the Pagesize value

**C.** the FeedRange value

**D.** a stream-based iterator

**ANSWER: C**

**Explanation:**

**the FeedRange value** is correct because the Azure Cosmos DB for NoSQL change feed pull model supports reading changes for a specific feed range. A feed range represents a logical portion of a container's partition-key range space. By obtaining the container's feed ranges and assigning separate ranges to different threads or workers, the application can read the change feed concurrently while each thread processes its own distinct range. This enables parallelization that scales with the container's physical partitioning and avoids having every thread read the same change-feed scope. Each worker creates and maintains its own change-feed iterator and continuation state for the feed range it owns, allowing reliable incremental processing and resumption after interruptions. The SDK documentation describes using feed ranges with change feed iterators to distribute processing in custom pull-model implementations. See Change feed pull model in Azure Cosmos DB for NoSQL and Container.GetFeedRangesAsync method.

**QUESTION NO: 22**

You have an Azure Cosmos DB for NoSQL account1 that is configured for automatic failover. The account1 account has a single read-write region in West US and a and a read region in East US.

You run the following PowerShell command.

```
Update-AzCosmosDBAccountFailoverPriority -ResourceGroupName "rg1" -Name "account1" -FailoverPolicy @("East US", "West US")
```

What is the effect of running the command?

**A.** A manual failover will occur.

**B.** The account will be unavailable to writes during the change.

**C.** The provisioned throughput for account1 will increase.

**D.** The account will be configured for multi-region writes.

**ANSWER: D**

**Explanation:**

The account will be configured for multi-region writes. The PowerShell command enables multiple write locations for the Azure Cosmos DB account. With this setting enabled, both the existing West US region and the East US region can accept write operations, rather than restricting writes to only the current read-write region. Azure Cosmos DB replicates writes across the account's configured regions, which can reduce write latency for applications deployed close to different regions and improves availability for write workloads.

Multi-region writes are an account-level capability. Enabling it does not require creating another account or moving data: the account's existing regional configuration is updated so the configured regions participate as writable locations. Applications can then direct requests to nearby preferred regions while Cosmos DB maintains replication among those regions. The account's automatic failover configuration remains relevant to regional resiliency, while multi-region writes allows write availability across the enabled write locations. Microsoft documents the `EnableMultipleWriteLocations` account setting in the Set-AzCosmosDBAccount PowerShell reference and describes the behavior and benefits of multi-region writes in Azure Cosmos DB for NoSQL.

**QUESTION NO: 23 - (HOTSPOT)**

You are creating a database in an Azure Cosmos DB Core (SQL) API account. The database will be used by an application that will provide users with the ability to share online posts. Users will also be able to submit comments on other users ' posts.

You need to store the data shown in the following table.

| Type | Description |
|---|---|
| Users | Information about a user who will use the application |
| Posts | Text of up to 1,000 characters that a user will share with other users |
| Comments | Text of up to 280 characters that users will submit as a comment on a post |
| Interests | Information about a user's interests |

The application has the following characteristics:

Users can submit an unlimited number of posts.

The average number of posts submitted by a user will be more than 1,000.

Posts can have an unlimited number of comments from different users.

The average number of comments per post will be 100, but many posts will exceed 1,000 comments.

Users will be limited to having a maximum of 20 interests.

For each of the following statements, select Yes if the statement is true. Otherwise, select No.

NOTE: Each correct selection is worth one point.

| Statements | Yes | No |
|---|---|---|
| If you embed the posts data into the users data instead of creating a separate document for each post, you will increase the write operation costs for new posts | ○ | ○ |
| If you embed the comments data into the posts data instead of creating a separate document for each comment you will increase the write operation costs for new comments | ○ | ○ |
| If you embed the interests data into the users data instead of creating a separate document for each interest, you will increase the read operation costs for displaying the users and their associated interests | ○ | ○ |

**ANSWER:**

**Explanation:**

The correct selections are Yes for embedding posts in a user item, Yes for embedding comments in a post item, and No for embedding interests in a user item. Azure Cosmos DB charges request units for writes partly according to the size of the item being written. An embedded post is added by updating the parent user document. Because each user can have an unlimited number of posts and is expected to average more than 1,000 posts, the user item continually grows. Each new post write must update that growing item, so its write cost is higher than creating a small, independent post document. This also helps avoid eventually approaching the 2-MB maximum size of a Cosmos DB item.

Comments have the same write-cost pattern. A comment embedded in its parent post requires an update to the post item. Posts can have an unlimited number of comments, with many exceeding 1,000, so the parent item grows substantially over time. Updating that larger item for each new comment increases the request-unit cost of the comment write. Microsoft documents that item size affects the RU charge for write operations in Optimize cost for reads and writes in Azure Cosmos DB.

Interests are different because the collection is strictly bounded at 20 values per user. Embedding this small, bounded set keeps the user and associated interests together in one item. The application can retrieve the user and all associated interests with a single point read of that item, rather than retrieving related interest records separately. Therefore, embedding interests does not increase the read operation cost for this display scenario. This follows Cosmos DB data-modeling guidance to embed related data that is accessed together when the embedded set is bounded, described in Modeling and partitioning data in Azure Cosmos DB for NoSQL.