DAIICT - End-sem Exam (Winter 2021-22)

IT457 – Cloud Computing

MM: 40 Time: 2 Hrs

Name: …………………………………..

Student ID: …………………………………..

Instructions:

1. Write your name and DAIICT ID in the fields mentioned above.
2. The exam consists of 6 questions. The marks for each question are mentioned against it.
3. The answer to the questions needs to be written in the space provided below each question. **No separate answer sheet is to be submitted.**
4. No clarifications can be asked about the questions in this exam. You can make necessary assumptions based on the course content covered in the lectures.
5. Use spare sheet provided to prepare your answer before neatly writing the final version under the question.

Q1. Describe the following two data consistency levels available in Azure CosmosDB. 2 x 3 = 6

1) Bounded staleness

Bounded staleness consistency guarantees that the reads may lag behind writes by at most *K* versions or prefixes of an item or *t* time-interval.

the "staleness" can be configured in two ways: number of versions *K* of the item by which the reads lag behind the writes, and the time interval *t*

Bounded staleness offers total global order except within the "staleness window."

Bounded staleness provides a stronger consistency guarantee than session, consistent-prefix, or eventual consistency.

Azure Cosmos DB accounts that are configured with bounded staleness consistency can associate any number of Azure regions with their Azure Cosmos DB account.

The cost of a read operation (in terms of RUs consumed) with bounded staleness is higher than session and eventual consistency, but the same as strong consistency.

2) Session consistency

session consistency is scoped to a client session.

Session consistency is ideal for all scenarios where a device or user session is involved since it guarantees monotonic reads, monotonic writes, and read your own writes (RYW) guarantees.

Session consistency provides predictable consistency for a session, and maximum read throughput while offering the lowest latency writes and reads.

Azure Cosmos DB accounts that are configured with session consistency can associate any number of Azure regions with their Azure Cosmos DB account.

The cost of a read operation (in terms of RUs consumed) with session consistency level is less than strong and bounded staleness, but more than eventual consistency.

3) Consistent prefix

Consistent prefix guarantees that in absence of any further writes, the replicas within the group eventually converge.

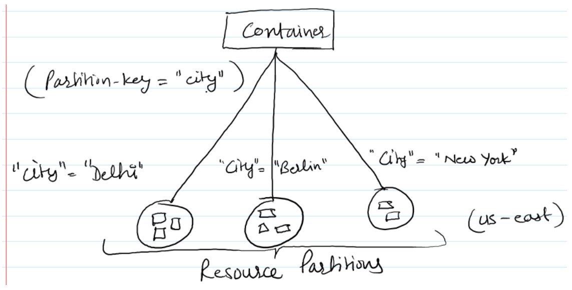
Consistent prefix guarantees that reads never see out of order writes. If writes were performed in the order A, B, C, then a client sees either A, A,B, or A,B,C, but never out of order like A,C or B,A,C.

Azure Cosmos DB accounts that are configured with consistent prefix consistency can associate any number of Azure regions with their Azure Cosmos DB account.

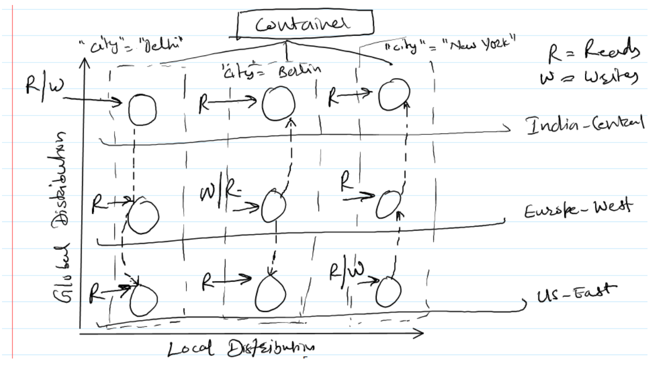
Q2. An Azure based web application provides information about various services by the municipality / local governments available for the residents of cities all over the world. Eg. If a user picks New Delhi as city, the app will list Health, Birth/Death Registration, Marriage Registration, Sanitary & Hygiene, Green spaces etc. Similarly, a person who is present in another country's city, say Berlin, will get list of government services provided by that city's governing body. Tapping on a service category will then provide more detailed information about that part of the service. Users can register for services in the app to avail the services further.

If this Azure web app uses CosmosDB as NoSQL database for its data regarding cities and their services, explain how would it partition its data horizontally and using these partitions how does it provide global distribution and consistency when data is written to (eg. When user registers for a service)? Explain your answer by suitable diagram(s) taking example of Berlin (Europe-west), Delhi (India-central), and New York (US-east) as examples of cities and the Azure data centers which carry their data.

7



Data inside a CosmosDB container is ***horizontally* partitioned based on the partition key**. In this example, the container will have 3 partitions for partition key "city" ie, one partition each for Delhi, Berlin and New York cities.



As the figure suggests, for global distribution, the **data of a container is available at resource partitions present in all three of the regions.** But, **only the local partition is written to** when the user registers for service or any update happens to a service data. Then, the **change gets update in partitions present in other regions eventually (eventual-consistency)**

Q3. What is a poison message in Azure ServiceBus Queue? How does it handle it and explain the role played by Dead Letter Queue in this handling?

5

In Azure ServiceBus Queue, the handlers code gets invoked when a message arrives. This invocation basically **instantiates the same code of handler each time a message comes**. Generally, in order to eliminate chances of loosing a message, the handler code **first handles the message and then removes from the queue**. But if there is a message that **crashes the handler code**, the **system is stuck in a loop** because such message will not get removed because the handler code crashed in between before it could remove the message from queue. The queue will again instantiate the handler to handle the same message because it is still present in the queue. This makes system stuck in a loop and choke because no messages can be further handled by the system. Such a message that causes this choking is called a poison message.

To handle such, ServiceBus Queue could simply **delete the message** after few tries. But this will **deprive chance for the developer to investigate handler code** as to why a particular message crashes the code. So, instead the ServiceBus **moves the poison message to a back-up queue, called the DeadLetter queue**. This queue is never exposed to public or used by system in production to process messages. The developer can routinely check the presence of poison messages in this queue and investigate independently, do the fixes in handler code and update it in production.

Q4. You have built a video portal where you allow sellers to sell their movies or other video content, customers to order, view and rate the movies. This web application has been built using the microservices architecture. As you can imagine, it consists of various microservices like UserManagement (for customer registration), SellerManagement (for sellers uploading content), Orders (for customers placing orders for video / movies), Feedback (for capturing customer reviews & ratings on various videos) etc.

In the portal, each video has its own page. In that page, there is a portion, called Reviews, that shows the reviews, ratings and customer names who did the ratings for that video. Lately, you are observing that this Reviews section is taking lot of time to load the complete information about the reviews for a movie.

Given that everything is up and running fine, what is the likely cause of this latency on the Reviews section of the page? What is the recommended pattern to solve it? Use the Reviews example to describe your solution.

7

In microservices architecture, each microservice has its own database. **One microservice does not query data from other microservice**. For any data that belongs to other microservice, it has to be queried through either the API or the microservice that owns the data or through other mechanisms like messaging.

The Reviews section is facing latency because of this reason – for showing all of Reviews data, **multiple microservices need to be queried**, like, UserManagement, Feedback, VideoCatalog and therefore, it will take more time to gather the data, put it in form of what is needed on the client side and then send it.

This problem is **solved by Materialized-View Pattern**. In this pattern, the data that is required on the client side from multiple microservices is put in a table that is directly accessible by a microservice or the controller/handler of the API. i.e. the requisite data is 'materialized' or serialized into the table. The data must meet two criteria for this pattern to be applied –

1) the data should be readonly,

2) it should be okay to have eventual consistency for that data.

Assuming that Reviews sections is a list of reviews where each review shows the content of the review, the 'star' rating, the customer name etc. the Materialized-View pattern is applicable because reviews are queries to the table (ie readonly) and it generally okay if very latest review is still not visible for some user (eventually consistency).

Solution:

The solution to the latency problem in the current question, therefore, is to create a table of all Reviews data required by the view in the same format as required on the front end so that there is no further processing required. In this case, the Review table will have columns like ProductID, Customer Name, Review Text, Star Rating etc. Each row of the table is the information shown for one review on a product. To show all reviews on a product we'll simply query by the ProductID to get all rows of the product whose page is in the view.

Updating the table: The Reviews table can be updated using a background task that could run at required frequency. This process would read the data from different microservices and update the Reviews table.

Q5. In the video portal mentioned in Q4, after login, the user is shown a page where following information specific to the logged in user is displayed:

- a horizontal list of cards/thumbnails showing currently trending video content in the country of the user. The card shows the name of the picture, and a photo of a scene. The user could simply click on one of the cards and the video will start playing. Apparently, every user of the app in a country will see the same list.
- a horizontal "Continue watching" list that shows cards of videos that the customer had started watching but did not finish. Each such card shows % completed video.

Design a NoSQL data solution using Azure CosmosDB Storage Tables that could serve as source of the content the user specific page is showing. Also explain how the database will be able to serve the purpose of being data source for the Homepage of a logged in user.

$\boxed{7}$

The key requirements for NoSQL db design for CosmosDB Table storage is to **design partition-key and row-key such that the data that is queried can be retrieved just based on partition and row keys**. If there is a query where the data is required from the 'value' part of the table, then it becomes an expensive operation. The database design should avoid that.

Table 1: Users

Partition-key: country

Row-key: userid

Values: JSON of all required properties, like Name, Subscription ID, profile [] etc.

We can assume that when the user is logged in, the system is able to detect the location of the user and, therefore, the country and can search for user's ID from the row key quickly to log the user in. Since the question does not require us to cover the login scenario we can assume that the login credentials are used to login the user. 2nd option could be for user to select the country first and then login.

Table 2: Trends

Partition-key: country

Row-key: trendRank

Value: json object of trends information. Movie name, URL of the thumbnail, URL of the video stream, and any other metadata information that needs to be shown to the user.

This table has partition-key as country. The row-key is not very significant here, we could use same row key for all the data, but it could be used as trendRank because we could then show the trending cards in decreasing order of trends.

Since the user is already logged in, the system is already aware of the country – ie partition-key value is already known. To get the latest trends to show to user would, therefore, be **very quick because the query on country and all rows in that partition will make the data available**. Then the back-end system can process the information in value field to provide stream of the thumbnail. Later, if the user clicks on the card, the video streaming URL is already known which will be used to stream the video.

Table 3: Watching

Partition-key: country

Row-key: userid

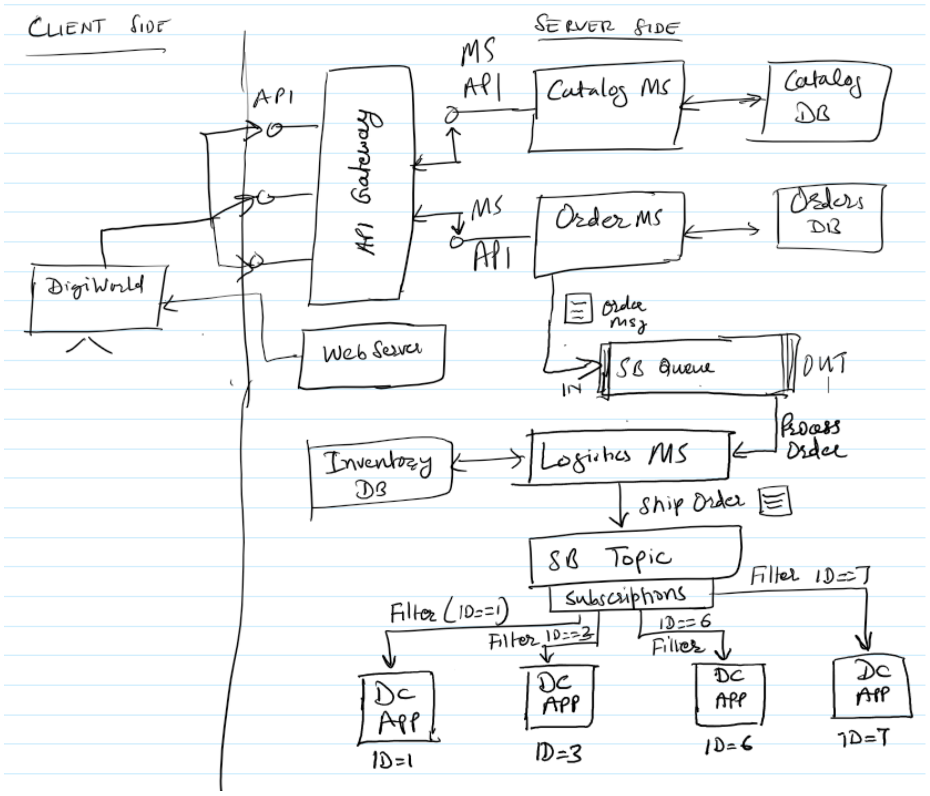
Value: JSON of collection of objects where each object represents a stream that the user was watch. This objet contains, in JSON, the details of the video – like, name, video URL, percent watched etc.

After the user logs in, both the country and user ID is available with the system. Therefore, it is very quick to query the content under "Continue Watching" view from Watching table. The back-end code will query the table on the country and user ID, get the collection of the objects and then return the data to client side for showing. When user clicks on any card, the URL for video stream is available to start the streaming. If the user stops watching, the information is updated to the table through ServiceBus messages to avoid blocking the UI when an object in the collection gets update in the value part (due to change of percentage) – this write operation will be slower because the collection will have to be parsed and one of the objects then needs to be updated.

Q6. Focusing on the functional aspects of the solution (ie ignoring the DB part), design a microservices based architecture for the following scenario.

The DigiWorld is an online portal that sells all kinds of electronics gadgets and appliances (like, Reliance Digital). It has a portal where users can see on Hompage the catalog of products among other UI elements. Then, the users can select a particular product and order it. To complete the order, user must provide shipping address with PIN. This PIN is used by the back-end to send the request of shipping the product to a DC (distribution center) closest to the PIN. For the DCs, DigiWorld has an application separate from the portal to manage the shipping of the items. This app is configured separately for each DC. Assuming DigiWorld has distribution centers in Delhi (for PIN starting with 1), Mumbai (for PIN starting with 3), Kolkata (for PIN starting with 7) and Chennai (for PIN starting with 6), each DC should receive orders meant for only that DC.

You should draw architecture diagram and provide key API, components, and cloud services to be used for enabling basic scenario of displaying a catalog in the homepage, ordering an item, and the DC relevant for ordered item receiving the order to ship. You can make assumptions necessary to explain/provide the solution architecture to relevant portions of the system.



The microservice architecture is shown above for this hypothetical system. Salient points –

1. Microservices: Catalog, Orders, Logistics (and other which are not depicted in the diagram) are microservices catering to one domain of application's functionality or a feature. Eg. Catalog service is responsible for providing the product information through its API. It will have API like GetCatalog.
    a. Each microservice has its **own database**. No microservice is can access database of other microservice directly.
    b. Microservices communicate with each other through either their exposed API or other mechanisms like messaging. Here, in current architecture, the Orders and Logistics microservice **communicate through Service Bus Queue**. The Orders microservice places message that has details of the customer order to the Queue. The Logistics microservice

is listener to the queue and as soon as a message arrives in the Queue, its callback function gets executed through which it can then process the order.

2. API Gateway: The API gateway **encapsulates the detailed API of each microservice** and provides to the external systems (clients) a uniform API based on same root URL. Thus the external system do not get to see internal details of the system (which Microservices are there and what is their API). This un-complicates clientside code by providing a uniform API. API gateway could be used for other functionalities like SSL termination.
3. Web Server: To display initial page on the client side browser, there is **need of a web server to serve pages like the homepage**. This server receives HTTP request when user types www.digiworld.com. It then serves the HTML, the required JavaScript and other files. The JavaScript code then gets executed (during page load etc. events) and calls the API of API gateway to get the data. The API gateway redirects the call to proper microservice to get the response and relays it back to client.
4. DC App integration through Service Bus Topic:
   Setup: Each DC runs same app, but configured to that DC. Let's assume that it is configured by different value of ID to each DC. This value is same as first digit of the PIN in the address. When a DC app is launched in the DC, it reads from its configuration file the value of ID, **registers itself to a subscription on the service bus topic with Filter ID value set to its ID**. Eg. for Delhi DC app, the filter condition will read ID==1.
   Communication: The Logistics app must set the ID value appropriately in the message that it is posting to the Topic. Eg. if the order is for Delhi DC, then **the message should have property "ID" set to "1".** Then when the Logistics app posts this message in the Topic, the topic automatically sends the message only to that DC app which matches the filter condition, the Delhi DC in current example, because it has filter set to ID == 1.