Microsoft

Module 04: Develop solutions that use Cosmos DB storage

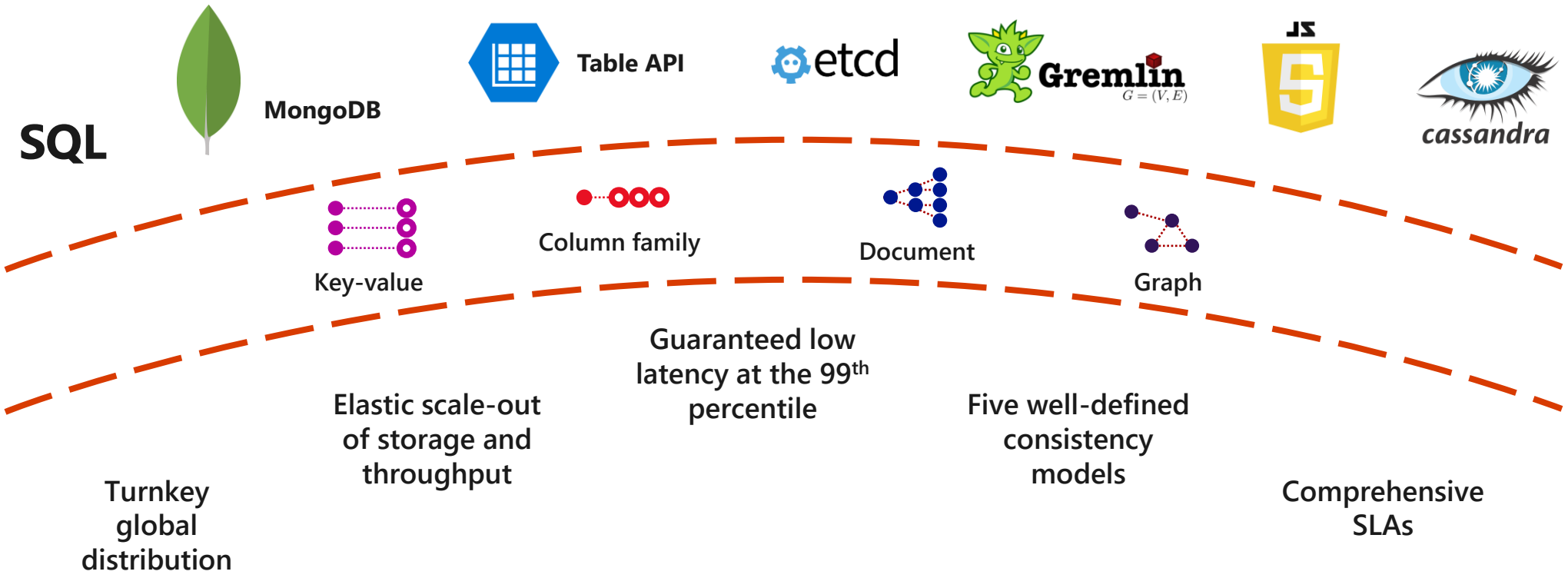
Microsoft

Topics

- Azure Cosmos DB overview
- Azure Cosmos DB data structure
- Create and update documents by using code

Lesson 01: Azure Cosmos DB overview

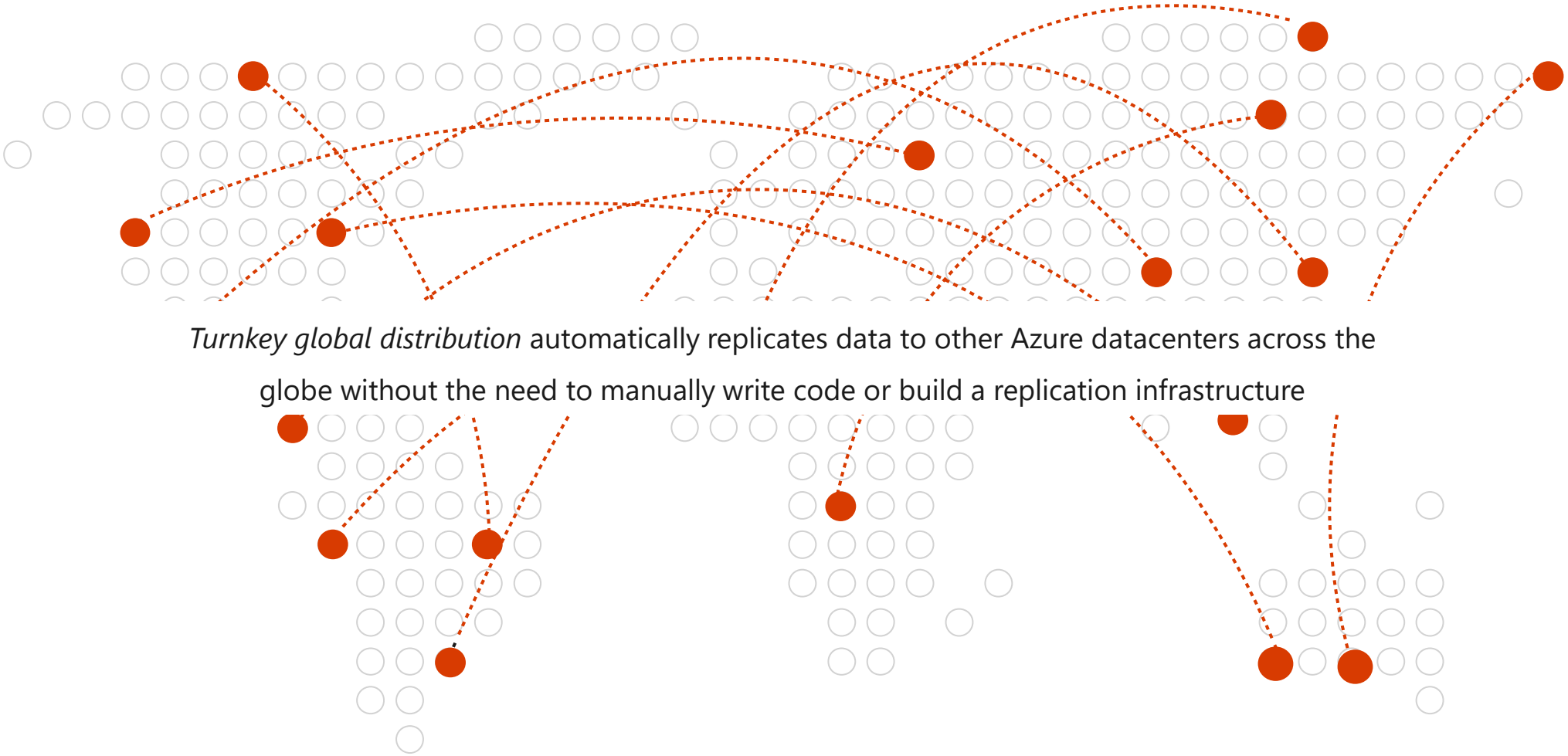
Azure Cosmos DB



Core functionality

- Global replication
  - Automatic and synchronous multi-region replication
  - Supports automatic and manual failover
- Varied consistency levels
  - Offers five consistency models
  - Provides control over performance-consistency tradeoffs, backed by comprehensive SLAs
- Low latency
  - Serve <10 ms read and <15 ms write requests at the 99th percentile
- Elastic scale-out
  - Elastically scale throughput from 10 to 100s of millions of requests/sec across multiple regions
  - Support for requests/sec for different workloads

Global Replication



Turnkey global distribution automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure

Request Units

- In Azure Cosmos DB, you provision throughput for your containers to run writes, reads, updates, and deletes.
- Azure Cosmos DB measures throughput using something called a request unit (RU).
- Request unit usage is measured per second, so the unit of measure is request units per second (RU/s)
  - A single request unit, one RU, is equal to the approximate cost of performing a single GET request on a 1-KB document using a document's ID
  - The number of request units consumed for an operation changes depending on the document size, the number of properties in the document, the operation being performed, and some additional concepts such as consistency and indexing policy

Partition Keys

- A partition key is the value by which Azure organizes your data into logical divisions
- A partition key defines the partition strategy, it's set when you create a container and can't be changed.
- A hot partition is a single partition that receives many more requests than the others, which can create a throughput bottleneck
- When you're trying to determine the right partition key and the solution isn't obvious, here are a few tips to keep in mind.
  - Don't be afraid of choosing a partition key that has a large number of values. The more values your partition key has, the more scalability you have.
  - To determine the best partition key for a read-heavy workload, review the top three to five queries you plan on using. The value most frequently included in the WHERE clause is a good candidate for the partition key.
  - For write-heavy workloads, you'll need to understand the transactional needs of your workload, because the partition key is the scope of multi-document transactions.

Request Unit considerations

While you estimate the number of RUs per second to provision, consider the following factors:

- Item size & Item indexing
- Item property count & Indexed properties:
- Data consistency: The strong and bounded staleness consistency levels consume approximately two times more RUs on read operations when compared to that of other relaxed consistency levels.
- Query patterns: The complexity of a query affects how many RUs are consumed for an operation. Factors that affect the cost of query operations include:
- Script usage: As with queries, stored procedures and triggers consume RUs based on the complexity of their operations. As you develop your application, inspect the request charge header to better understand how much RU capacity each operation consumes.

Consistency levels

Azure Cosmos DB provides five consistency levels:

Strong

The staleness window is equivalent to zero, and the clients are guaranteed to read the latest committed value of the write operation.

Bounded staleness

Clients always read the value of a previous write, with a lag bounded by the staleness window.

Consistency levels (continued)

Consistency Level	Description
Strong	When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.
Bounded Stateless	This level is similar to the Strong level with the major difference that you can configure how stale documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
Session	This level guarantees that all read and write operations are consistent within a user session. Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.
Consistent Prefix	This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
Eventual	This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary. This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize it's transactions.

APIs

- MongoDB API
  - Acts as a massively scalable MongoDB service powered by the Azure Cosmos DB platform
  - Compatible with existing MongoDB libraries, drivers, tools, and applications

- Table API
  - A key-value database service built to provide premium capabilities to existing Azure Table storage applications without making any app changes

- Gremlin API
  - A fully managed, horizontally scalable graph database service
  - Easy-to-build and run applications that work with highly connected datasets supporting Open Graph APIs (based on the Apache TinkerPop specification, Apache Gremlin)

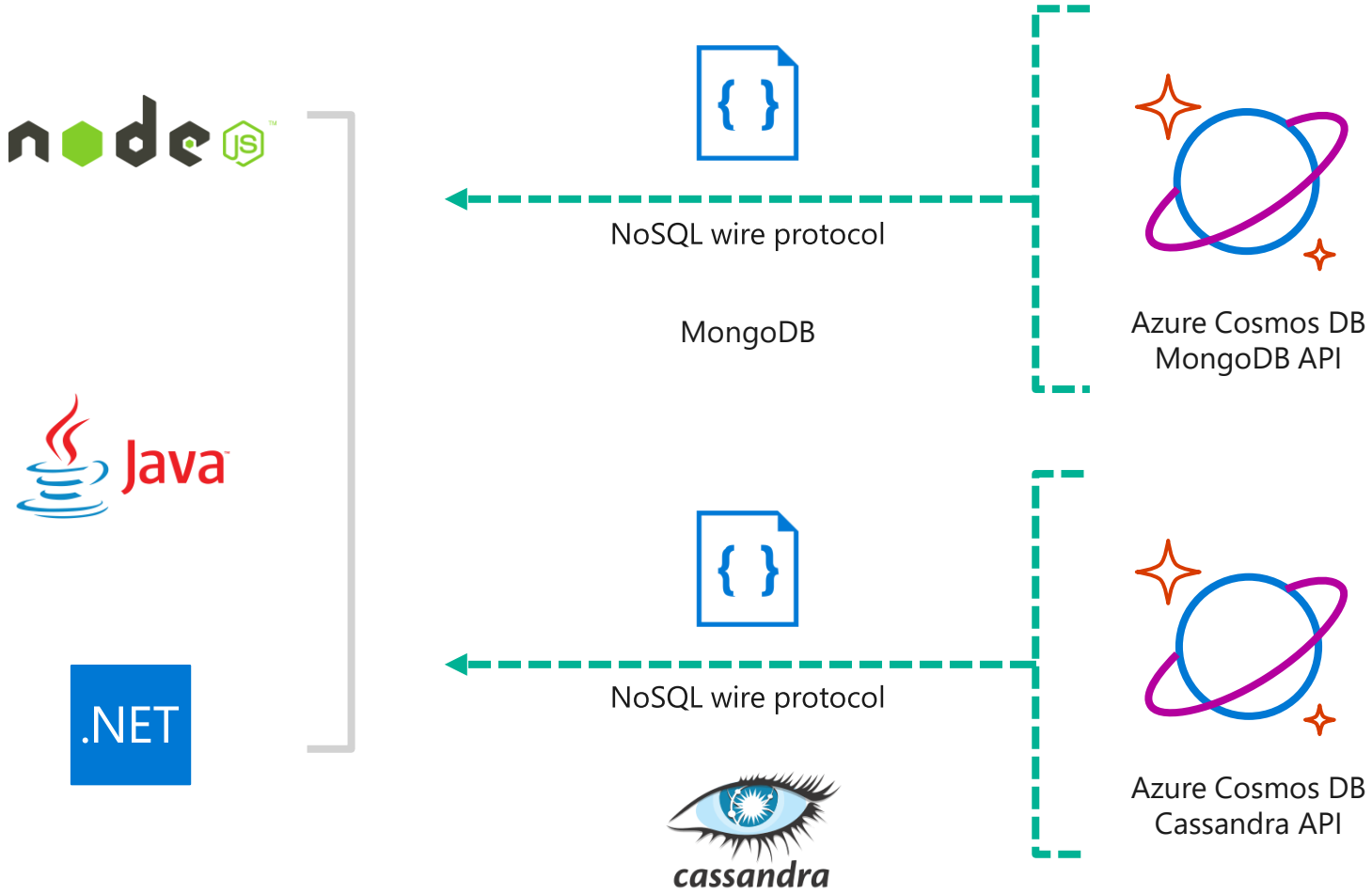
APIs (cont.)

- Cassandra API
  - Globally distributed Apache Cassandra service powered by the Azure Cosmos DB platform
  - Compatible with existing Apache Cassandra libraries, drivers, tools, and applications
- SQL API
  - JavaScript and JavaScript Object Notation (JSON) native API based on the Azure Cosmos DB database engine
  - Provides query capabilities rooted in SQL
  - Query for documents based on their identifiers or make deeper queries based on properties of the document, complex objects, or the existence of specific properties
  - Supports the execution of JavaScript logic within the database in the form of stored procedures, triggers, and user-defined functions

Migrating from NoSQL

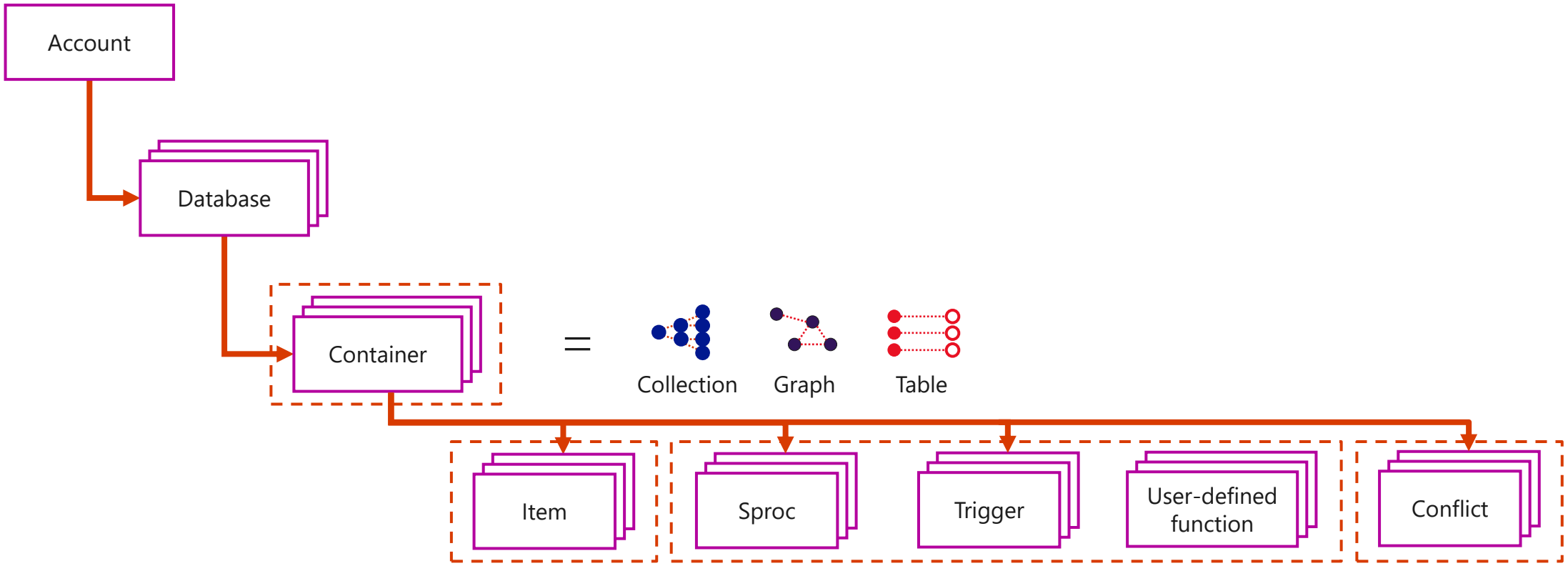
- Many NoSQL database engines are simple to get started with, but they might cause problems as you scale, including:
  - Tedious setup and maintenance requirements for a multiple-server database cluster
  - Expensive and complex high-availability solutions
  - Challenges in achieving end-to-end security, including encryption at rest and in flight
  - Required resource overprovisioning and unpredictable costs to achieve scale
- Azure Cosmos DB provides NoSQL-as-a-service for:
  - MongoDB
  - Cassandra
  - Gremlin

Migrating from NoSQL (continued)



Lesson 02: Azure Cosmos DB data structure

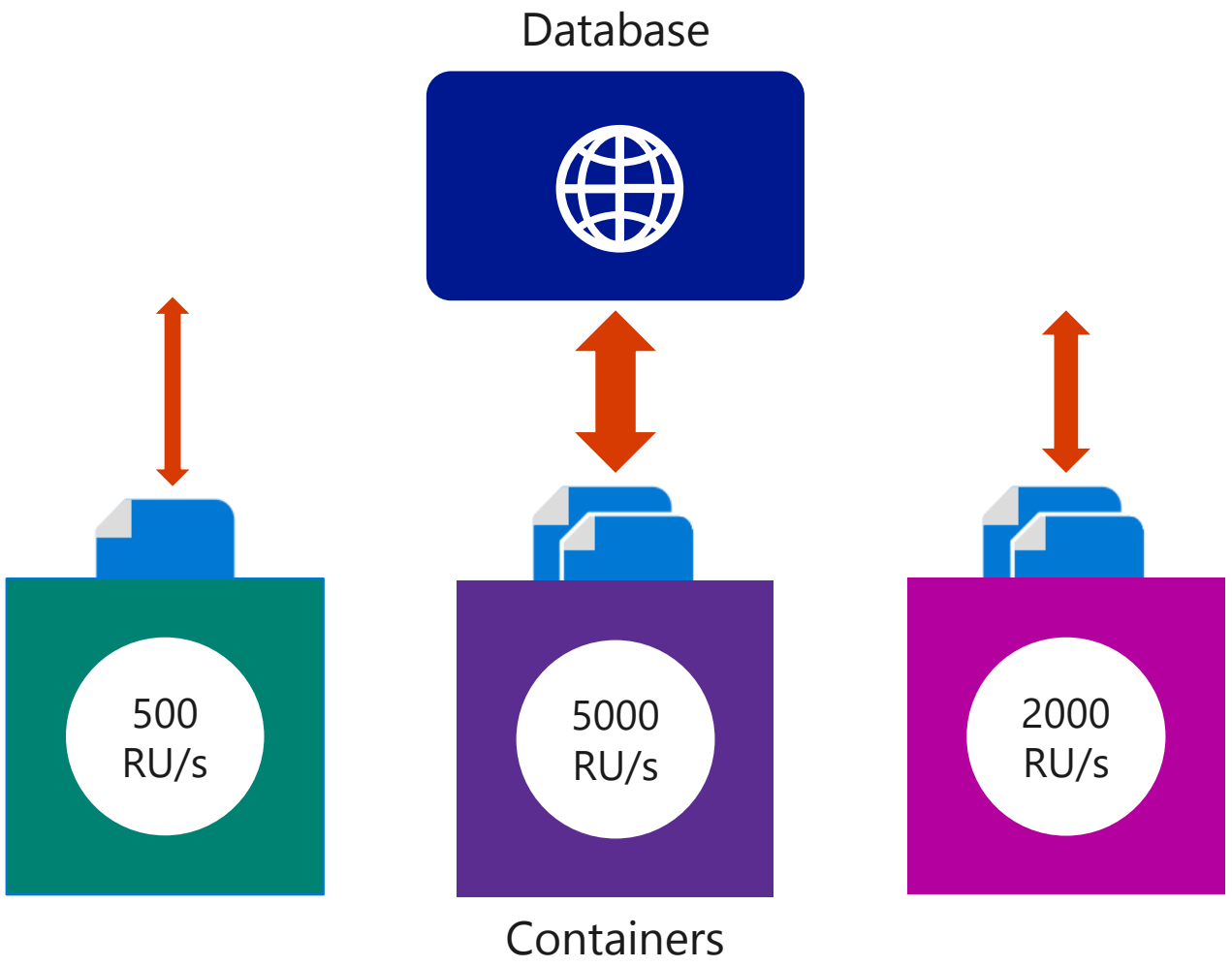
Resource hierarchy



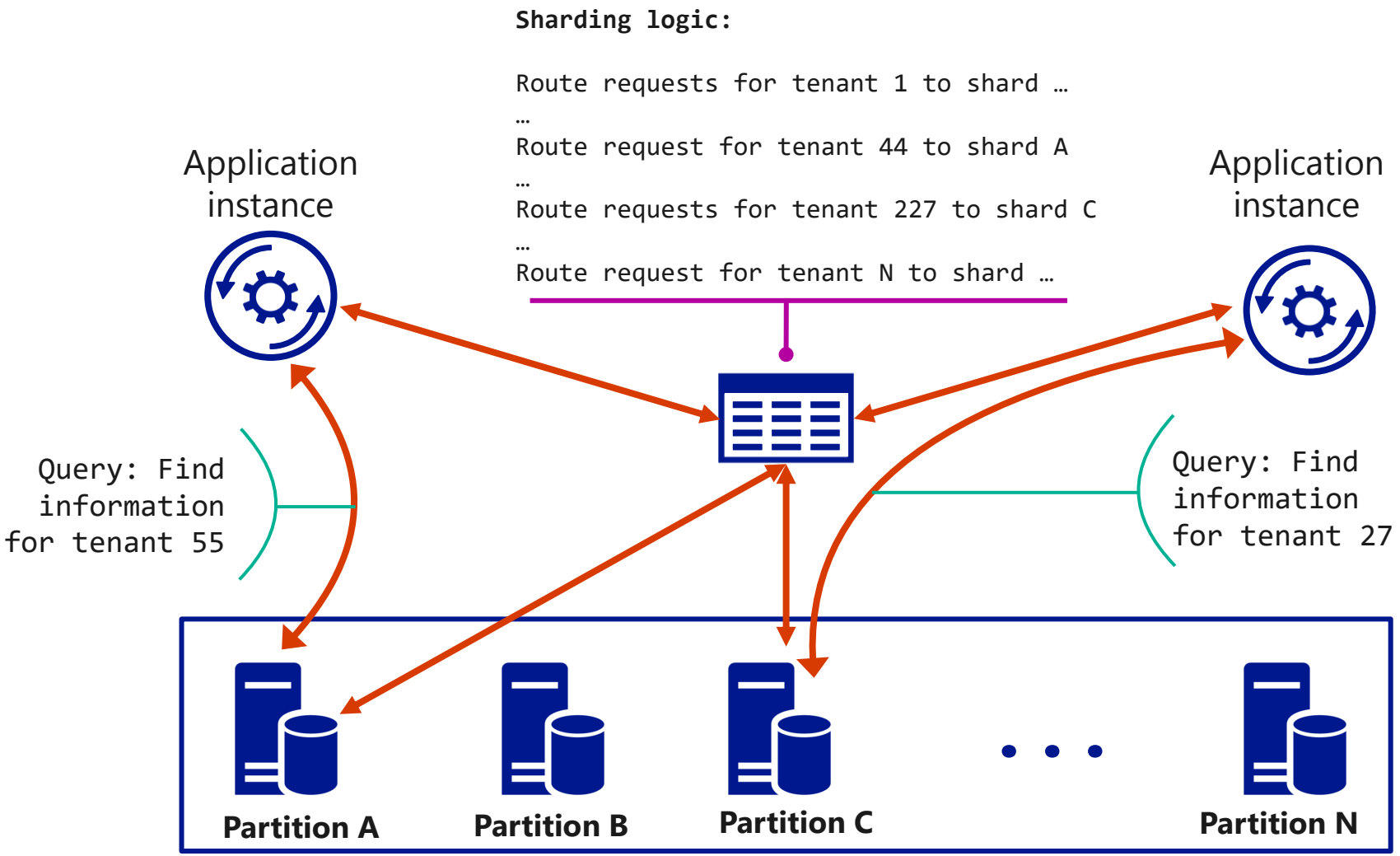
Resource hierarchy (continued)

Resource	Description
Account	A set of databases
Database	Logical container for containers that can (optionally) share throughput across the containers
Collection (container)	A group of Items and programmatic resources usually related in some way
Document (item)	An arbitrary unit of content In many cases, this would be a JSON document
Stored procedure (sproc)	Application logic written in JavaScript executed within the database engine as a transaction
Trigger	Application logic written in JavaScript executed before or after either an insert, replace, or delete operation
User-defined function	Application logic written in JavaScript to extend the SQL API query language

Containers



Partitioning

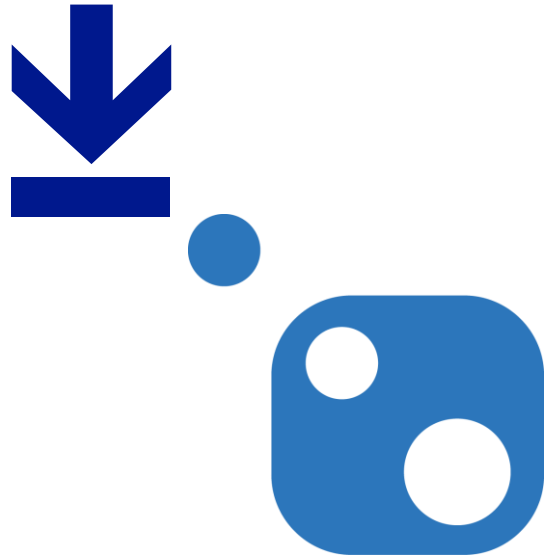


Demonstration: Create Azure Cosmos DB resources by using the Azure Portal

Lesson 03: Create and update documents by using code

Manage collections and documents

- Install the **Microsoft.Azure.Cosmos** package from NuGet:
  - `dotnet add package Microsoft.Azure.Cosmos`
- Use the following namespaces:
  - `Microsoft.Azure.Cosmos`
  - `Microsoft.Azure.Cosmos.Linq`
- Use the **CosmosClient** class

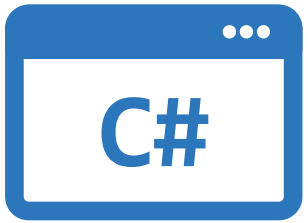


Creating a CosmosClient instance by using .NET

```
using Microsoft.Azure.Cosmos;
using Microsoft.Azure.Cosmos.Linq;

string endpoint = "[endpoint]";
string key = "[key]";

CosmosClient client = new CosmosClient(endpoint, key);

AccountProperties account = await client.ReadAccountAsync();
```

Obtained from the account

Read account configuration



Accessing a database by using .NET

```
CosmosClient client = new CosmosClient(endpoint, key);

string databaseName = "DemoDatabase";
```

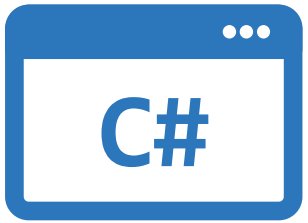
```
Database database = client
    .GetDatabase(databaseName);
```

Reference existing database

OR

```
Database database = await client
    .CreateDatabaseIfNotExistsAsync(
        databaseName,
        throughput: 10000
    );
```

Create new database



Accessing a collection by using .NET

```
CosmosClient client = new CosmosClient(endpoint, key);
Database database = client.GetDatabase(databaseName);
string collectionName = "ExampleCollection";
```

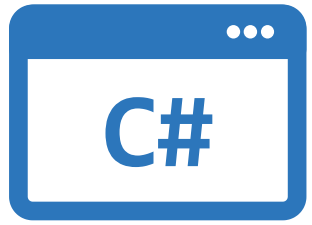
```
Container container = database
    .GetContainer(collectionName);
```

Reference existing container

OR

```
Container container = await database
    .CreateContainerIfNotExistsAsync(
        containerName, partitionKey,
        throughput: 400
    );
```

Create new container



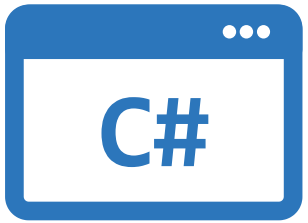
Demonstration: Managing Azure Cosmos DB by using .NET

Creating documents by using .NET

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName, collectionName);

// create anonymous type in .NET
Product orangeSoda = new Product {
    id = "7cc3212d-0e2c-4a13-b348-f2d879c43342",
    name = "Orange Soda", group = "Beverages",
    diet = false, price = 1.50m, quantity = 2000
};

// Upload document
Product item = await container.CreateItemAsync(orangeSoda);

Product item = await container.UpsertItemAsync(orangeSoda);
```

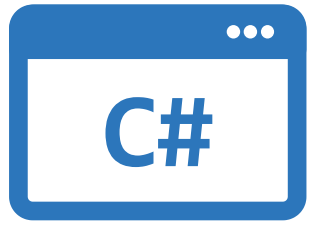
Create new document

Create or replace document



Reading documents by using .NET

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName, collectionName);

// Get unique fields
string id = "7cc3212d-0e2c-4a13-b348-f2d879c43342";
PartitionKey partitionKey = new PartitionKey("Beverages");

// Read document using unique id
ItemResponse<Product> response = await container.ReadItemAsync<Product>(
        id,
        partitionKey
);

// Serialize response
Product item = response.Resource;
```

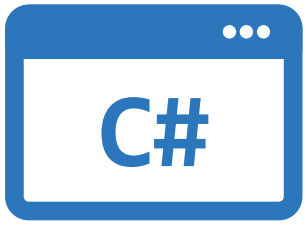


Querying documents by using .NET

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName, collectionName);

// Use SQL query language
FeedIterator<Product> iteratorOld = container.GetItemQueryIterator<Product>(
    "SELECT * FROM products p WHERE p.diet = false"
);

// Iterate over results
while (iterator.HasMoreResults)
{
    FeedResponse<Product> batch = await iterator.ReadNextAsync();
    foreach(Product item in batch)
    { }
}
```

Check for new batch of results

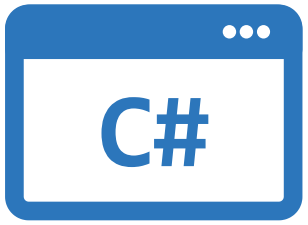
Get next batch of results



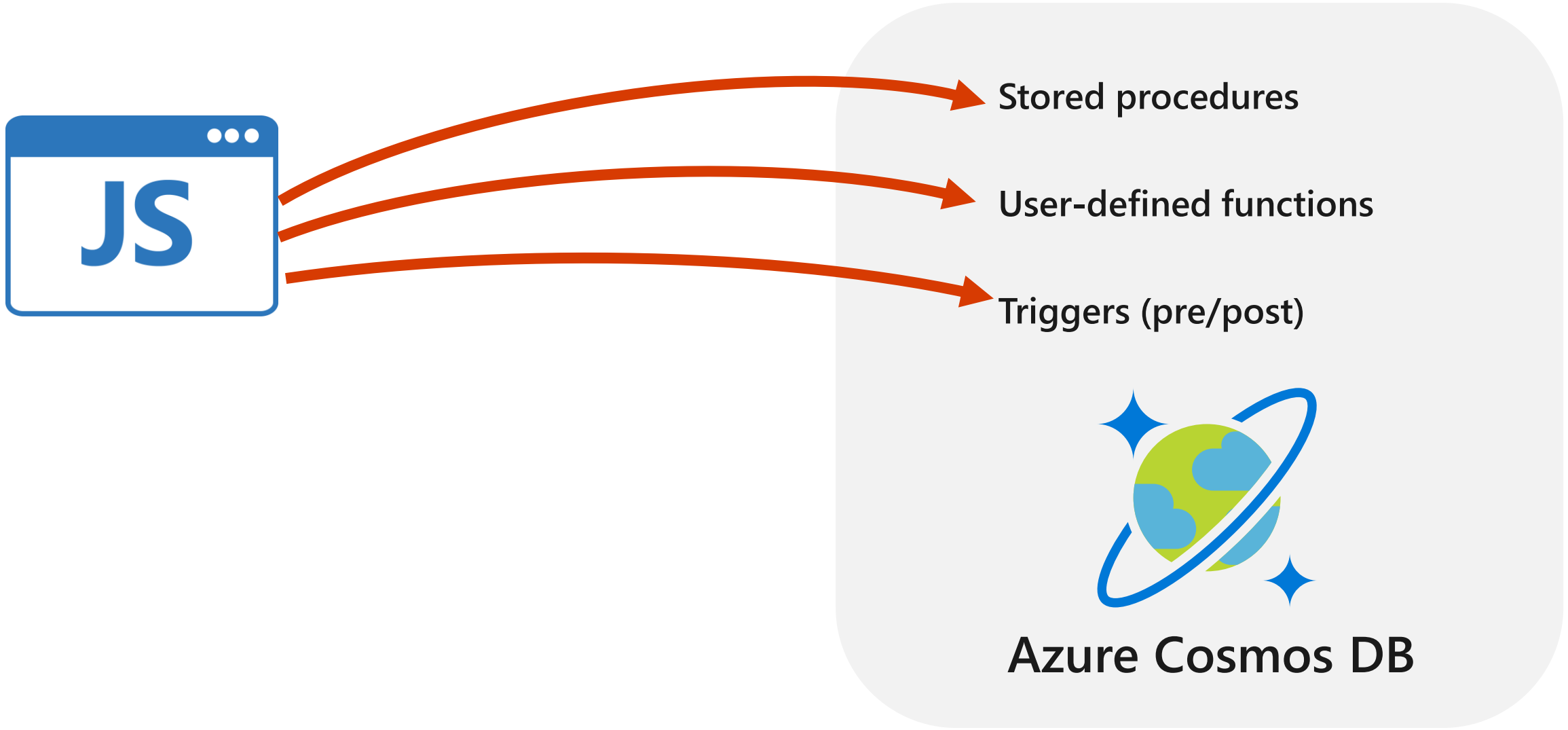
Querying documents by using .NET (continued)

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName, collectionName);

// Use LINQ query language
FeedIterator<Product> iterator = container.GetItemLinqQueryable<Product>()
    .Where(p => !p.diet)
    .ToFeedIterator();

// Iterate over results
while (iterator.HasMoreResults)
{
    FeedResponse<Product> batch = await iterator.ReadNextAsync();
    foreach(Product item in batch)
    { }
}
```

Translate LINQ expression to SQL

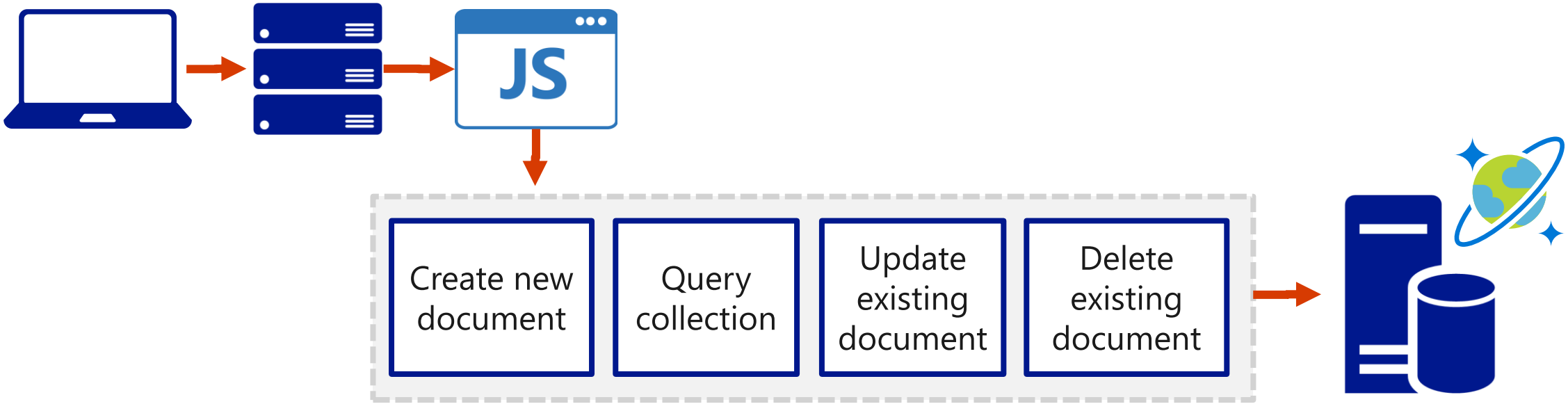


JavaScript and Azure Cosmos DB



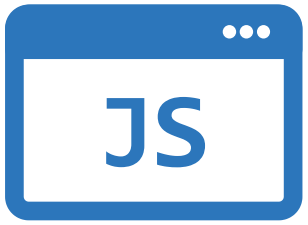
Stored procedures

- In Azure Cosmos DB, JavaScript is hosted in the same memory space as the database
- Requests made within stored procedures and triggers run in the same scope of a database session



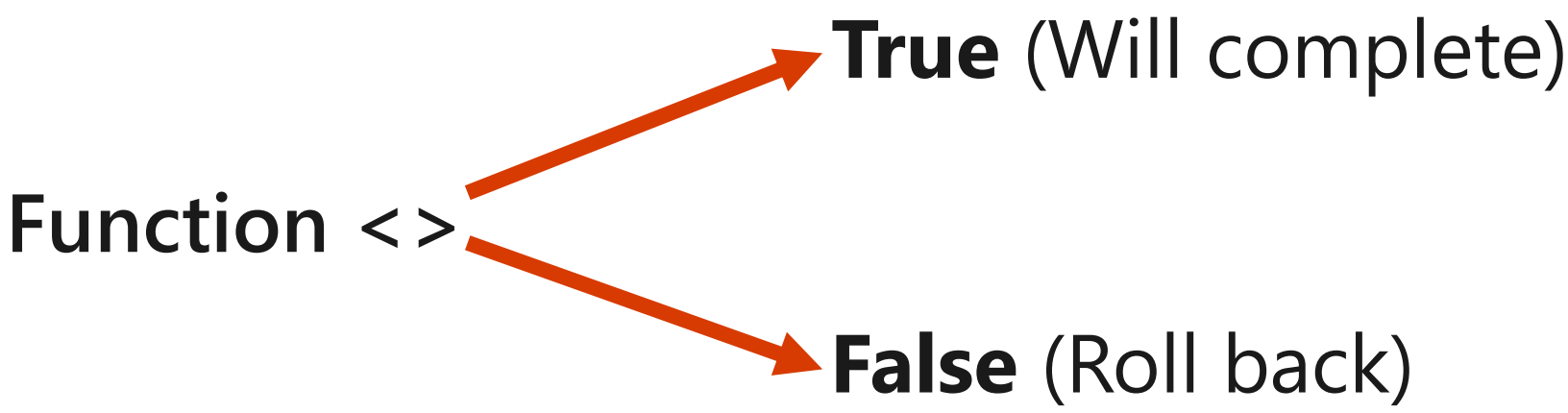
Stored procedure in JavaScript

```
function createSampleDocument(documentToCreate) {
    var context = getContext();
    var collection = context.getCollection();
    var accepted = collection.createDocument(
        collection.getSelfLink(),
        documentToCreate,
        function (error, documentCreated) {
            context.getResponse().setBody(documentCreated.id)
        }
    );
    if (!accepted) return;
}
```



Bounded execution

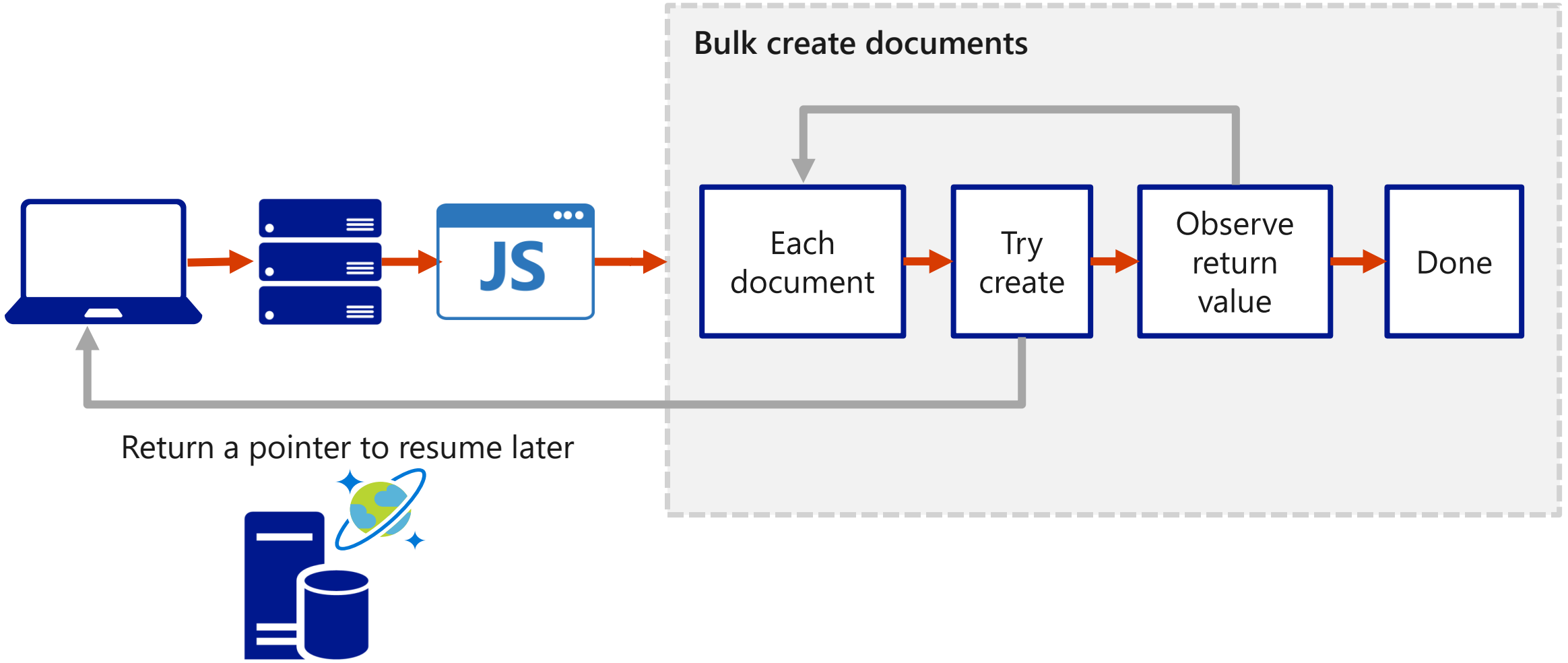
- All Azure Cosmos DB operations must complete within a limited amount of time
  - Specifically, stored procedures have a limited amount of time to run on the server
- All collection functions return a Boolean value that represents whether that operation will complete or not



Transaction continuation

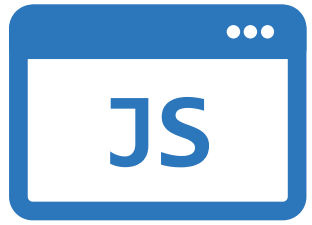
- JavaScript functions can implement a continuation-based model to batch or resume execution
- The continuation value can be any value of your choice
- Your applications can then use this value to resume a transaction from a new starting point

Transaction continuation (cont.)



User-defined functions in JavaScript

```
var taxUdf = {
    id: "tax",
    serverScript: function tax(income) {
        if (income == undefined)
            throw 'no input';
        if (income < 1000)
            return income * 0.1;
        else if (income < 10000)
            return income * 0.2;
        else
            return income * 0.4;
    }
}
```

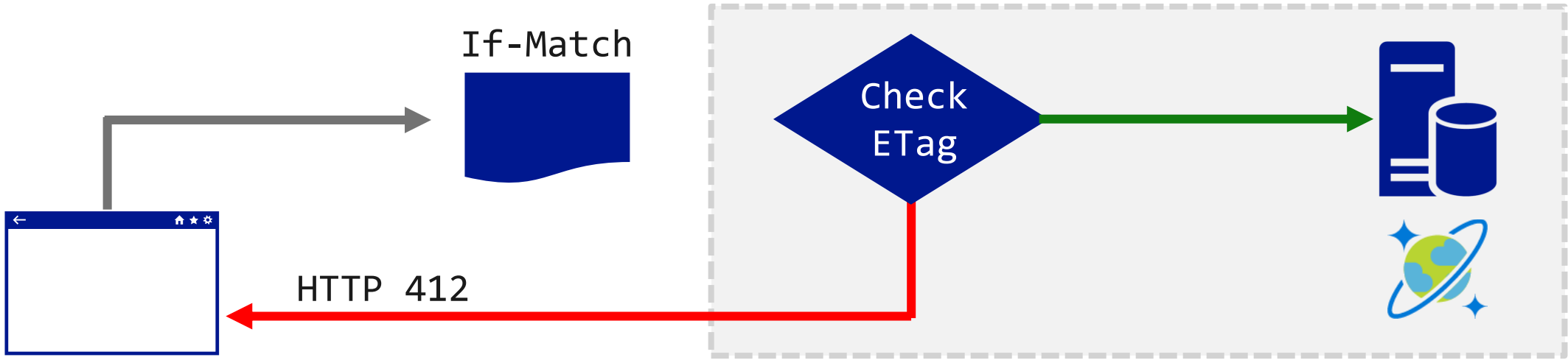


User-defined functions in SQL queries

```
SELECT
    *
FROM
    TaxPayers t
WHERE
    udf.tax(t.income) > 20000
```

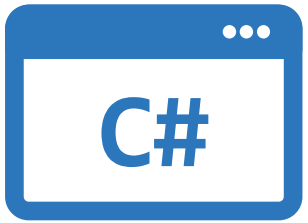
Optimistic concurrency

- The SQL API supports optimistic concurrency control through HTTP ETags
- Every SQL API resource has an ETag system property
- ETags can be used with the If-Match HTTP request header to allow the server to decide whether a resource should be updated



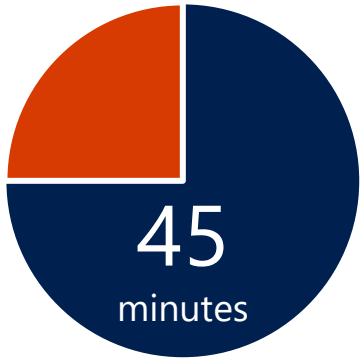
Controlling concurrency in .NET

```
try
{
    var ac = new AccessCondition { Condition = readDoc.ETag, Type =
        AccessConditionType.IfMatch };
    await client.ReplaceDocumentAsync(readDoc, new RequestOptions {
        AccessCondition = ac });
}
catch (DocumentClientException dce)
{
    if (dce.StatusCode == HttpStatusCode.PreconditionFailed)
    {
        Console.WriteLine("Another process has updated the record");
    }
}
```



Lab: Constructing a polyglot data solution

Duration



Lab sign-in information