Guide to NoSQL with Azure Cosmos DB

Work with the massively scalable Azure database service with JSON, C#, LINQ, and .NET Core 2

Packt>

www.packt.com

Gastón C. Hillar and Daron Yöndem

Guide to NoSQL with Azure Cosmos DB

Work with the massively scalable Azure database service with JSON, C#, LINQ, and .NET Core 2

Gastón C. Hillar
Daron Yöndem

BIRMINGHAM - MUMBAI

Guide to NoSQL with Azure Cosmos DB

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Reshma Raman
Content Development Editor: Chris D'cruz
Technical Editor: Dinesh Pawar
Copy Editor: Safis Editing
Project Coordinator: Nidhi Joshi
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Jisha Chirayil
Production Coordinator: Shantanu Zagade

First published: September 2018

Production reference: 1270918

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78961-289-9

www.packt.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packt.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Gastón C. Hillar is Italian and has been working with computers since he was 8 years old. Gaston has a bachelor's degree in computer science (graduated with honors) and an MBA. He is an independent consultant, a freelance author, and a speaker.

He was a senior contributing editor at Dr. Dobb's Journal and has written more than a hundred articles on software development topics. He has received the prestigious Intel Black Belt Software Developer award eight times.

He lives with his wife, Vanesa, and his two sons, Kevin and Brandon.

Daron Yöndem has been a Microsoft Regional Director and a Microsoft MVP for 11 years. He is a regular speaker at international conferences, recently focusing on microservices, serverless, DevOps, and IoT. Daron currently works as a CTO at XOMNI Inc, a cloud company that builds PaaS offerings for retailers, such as XOGO, who are building decision signage platform. Feel free to reach out to him on Twitter `@daronyondem`.

About the reviewers

Jim O'Neil is a senior architect at BlueMetal, a division of Insight, where he focuses on cloud architectures and IoT solutions for a wide array of industries. A former Microsoft Developer Evangelist and a Microsoft Azure MVP since 2017, he is a frequent speaker at software events in the greater New England area and has perennially organized and assisted at local conferences, such as Global Azure Bootcamp, Boston Code Camp, and New England GiveCamp. In his spare time, he has developed a penchant for genetic genealogy and has helped a number of adoptees (including himself!) find their birth families.

Roberto Freato has been an independent IT consultant since he started his professional career. While working for small software factories while he was studying for an MSc in computer science engineering, for which he produced a thesis about consumer cloud computing, he began to specialize in cloud computing and Azure. Today, he works as a freelance consultant for important companies in Italy, helping clients to design and launch their distributed software solutions. He trains for the developer community in his free time, speaking in many conferences. He has been a Microsoft MVP since 2010.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book shows you how to develop applications that work with Azure Cosmos DB. Azure and other cloud applications typically work with massive amounts of data that can be organized in different ways. These applications will often require elastic scalability of storage and throughput, and will often need to work across new geographical regions. This is the problem that Microsoft's Azure Cosmos DB service addresses. It is a globally distributed, massively scalable, and multi-model NoSQL database service.

You will learn how to use the Azure Cosmos DB Emulator. You will start by writing simple queries against Cosmos DB data and handling the responses, and then learn how to use more sophisticated querying constructs. You will create a full C# application that integrates with Cosmos DB, and learn about the .NET Core 2 classes that are needed to do so. You will work with LINQ and POCOs to cement your querying capabilities.

Having mastered Cosmos DB's NoSQL capabilities by the end of this book, you will be able to build scalable, globally distributed, and highly responsive applications.

Who this book is for

This book is for C# developers. You do not require any knowledge of Azure Cosmos DB, but familiarity with the Azure platform would be an advantage.

What this book covers

`Chapter 1`, *Introduction to NoSQL in Cosmos DB*, introduces you to the features of Cosmos DB and the NoSQL data model. You will also learn about the different elements of the Cosmos DB resource model, which will allow you to have a clear understanding of how to work with this database service.

`Chapter 2`, *Getting Started with Cosmos DB Development and NoSQL Document Databases*, will teach you how to provision a Cosmos DB account that uses the SQL API, and you will start working with a document database, along with its collections and documents. You will use the web-based Azure portal and work with Azure Cosmos DB Explorer.

`Chapter 3`, *Writing and Running Queries on NoSQL Document Databases*, will have you writing and running queries to retrieve data from the documents in a collection. You will learn how to use the Cosmos DB dialect of SQL to work against a document database with the SQL API. Most importantly, you will understand the different ways of working with documents, their sub-documents, and their arrays, and you will learn about how queries consume resource units.

`Chapter 4`, *Building an Application with C#, Cosmos DB, a NoSQL Document Database*, and the SQL API, will get you using Cosmos DB, the .NET Core SDK, the SQL API, and C# to code our first Cosmos DB application. The main focus will be on learning about many important aspects related to the SDK, as well as how to easily build a first version of the application we'll be working with.

`Chapter 5`, *Working with POCOs, LINQ, and a NoSQL Document Database*, will keep you working with the .NET Core SDK, but this time, you will work with POCOs and LINQ queries. You will take advantage of the strongly typed features of C# and the functional programming features that LINQ provides for working with Cosmos DB.

`Chapter 6`, *Tuning and Managing Scalability with Cosmos DB*, explores aspects that will enable you to design and maintain scalable architectures with Cosmos DB. You will use the sample application you will have built to understand complex topics related to scalability.

To get the most out of this book

You will need a Microsoft Azure account in order to use Cosmos DB. You should also be familiar with the Azure platform to get the most out of this book.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Guide-to-NoSQL-with-Azure-Cosmos-DB`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789612899_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Right-click on the `Documents` element for the `VideoGames1` collection."

A block of code is set as follows:

```
SELECT *
FROM Videogames v
WHERE v.id = '2'
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click **Create a resource** | **Databases** | **Azure Cosmos DB**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packt.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packt.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Introduction to NoSQL in Cosmos DB

In this chapter, we will start our journey toward developing applications that work with a globally distributed, massively scalable, and multi-model database service provided by Microsoft: Azure Cosmos DB. We will focus on a high-level technical overview of this innovative database service.

Modern applications that take advantage of Azure and other cloud platforms usually require working with massive amounts of data that might be organized in different ways. In addition, these applications require elastic scale out of storage and throughput. We might start with a few gigabytes, but we can end up with many petabytes in months. Our application can start working with most clients in California, but it might expand its clients in Germany, Switzerland, and Norway in the near future. Of course, our application will be continuously evolving and we will have to store more data related to each performed operation based on the new requirements. In this chapter, we will understand why Cosmos DB is an excellent candidate to be used as a database service in these kinds of applications.

In this chapter, we will cover the following:

- Making the paradigm shift to the NoSQL way
- Learning about the main features of Cosmos DB
- Understanding the supported NoSQL data models
- Using the appropriate API for each data model
- Diving deep into the Cosmos DB resource model
- Understanding the system topology
- Learning about the resource hierarchy for each container

Making the paradigm shift to the NoSQL way

During the last decade or so, the most popular databases have been relational database management systems. Hence, there is a huge number of developers who know how to build an application that requires the persisting and querying of data by creating tables and relationships in relational databases, such as Microsoft SQL Server.

However, when we work with C# and .NET Core, we work with object-oriented programming. LINQ makes it possible to easily query objects by adding functional programming features to C#, but we need to add a complexity layer between our application and the relational databases: an **Object-Relational Mapping** (**ORM**) solution, such as Entity Framework or NHibernate.

We have entities in our C# application, and the ORM maps these entities' relationships to the tables. This way, we can create an instance of an entity and persist it in the underlying tables in the relational database system. The ORM translates the operations into the necessary SQL for the relational database system to insert new rows in the appropriate tables.

Of course, we could continue explaining the different operations and how the objects in our application, the ORM, and the relational database make them happen. However, our goal is to start making the paradigm shift between the usual way of working (with relational databases) and the new way of working (with NoSQL databases).

The first version of an application that works with C#, an ORM, and a relational database is not a problem. However, when new requirements arrive and we need to add new properties to an existing object or relate it to another object, we have to perform migrations to make it possible to use the new objects in the different operations and persist them in the underlying relational database. We have to make changes in different places. We need to edit the classes that define the properties that an entity has to persist, we have to make sure that the ORM mappings are updated, and we need to ensure that the underlying relational database has the new columns in the necessary tables. Hence, we make changes in the code, in the ORM, and in the underlying database schema.

Whenever it is necessary to deploy a new version of the application, we have to make sure that the migration process is executed and that the underlying database schema has the required version for the C# code and the ORM configuration. The migration process makes the necessary changes in the tables and relationships to make it match the ORM mappings. Hence, a single property that needs to be added to an object and needs to be persisted generates a cascade of changes in different parts of our application. Of course, there are many ways of automating the necessary tasks. However, the fact that the tasks are automated doesn't mean that they aren't required.

Now, let's start thinking about the way in which we are going to work with a Cosmos DB NoSQL document database. We can start writing our first version of an application with C# and .NET Core, work with object-oriented programming, and use the provided methods in the Cosmos DB .NET Core SDK to persist the created objects in the schema-agnostic document database. There is no ORM between our application code and the NoSQL database service. We work with objects, we persist them, we retrieve them, and we query them. We only need to specify serialization and deserialization settings if necessary, but we don't have to worry about mapping an object to tables and their relationships.

Documents are objects.

Now, seriously, what else do we need to do to create our first version of the application? We have to learn how to work with the Cosmos DB .NET Core SDK, as well as the necessary tools for interacting with and managing a Cosmos DB database, in order to be ready for the first version of the application. We also have to understand the SQL dialect, which allows us to work against a Cosmos DB document database, in addition to many scalability and provisioning strategies.

A NoSQL database makes it easier to start working with a first version of an application compared to the process required with the traditional ORM and relational database management system combination. We work with documents, we store documents, we retrieve documents, we query documents. We don't require complex mappings and translations. We can work with object-oriented code without adding complex middleware such as an ORM.

Whenever it is necessary to deploy a new version of an application that is working against a Cosmos DB NoSQL document database, we don't have to worry about migration processes. If we need to add a single property to an object, we just add it and persist it in the schema-agnostic document database. There is no need to run any script that makes changes to the existing documents. We can continue working with the documents with a different schema, as they will be able to coexist with the new documents that have a new schema.

What about queries that work only with the new property? No problem—we can use properties or keys that don't exist in all the persisted documents; the schema-agnostic features support this scenario. For example, we can start running queries that check whether the value of the new property matches some specific criteria after persisting an object that has the new property.

You might be wondering, "why haven't I been working with NoSQL for the last 10 years?" There is a simple answer to this question: storage costs were higher and relational database management systems made it easy to optimize storage use while providing great database features. However, things have changed, and nowadays, we have new options. Cosmos DB provides a NoSQL database service that allows us to get up and running very quickly. We will learn how to create a first version and a second version of an application to simplify the paradigm shift to the NoSQL way of working with Cosmos DB.

Obviously, as always happens, relational databases will still be great for thousands of scenarios. However, be sure that the time you invest in learning Cosmos DB features will allow you to use its services in an application in which you thought that the only choice was a traditional relational database.

Learning about the main features of Cosmos DB

Cosmos DB extends the database service that was known as Azure Document DB. However, it is very important to note that Cosmos DB adds a huge number of features to the services offered by its predecessor. In fact, Cosmos DB is continuously adding new features and has quickly become one of the most innovative services found in Azure that targets mission-critical applications at a global scale.

Cosmos DB is a NoSQL database service included in Azure. NoSQL definitely means *not only SQL* in the case of this database service, because Cosmos DB provides a SQL API that allows us to query documents by using SQL in one of the possible models that the database service supports. Cosmos DB is a multi-model database service, and therefore it supports different non-relational models, which we will analyze later.

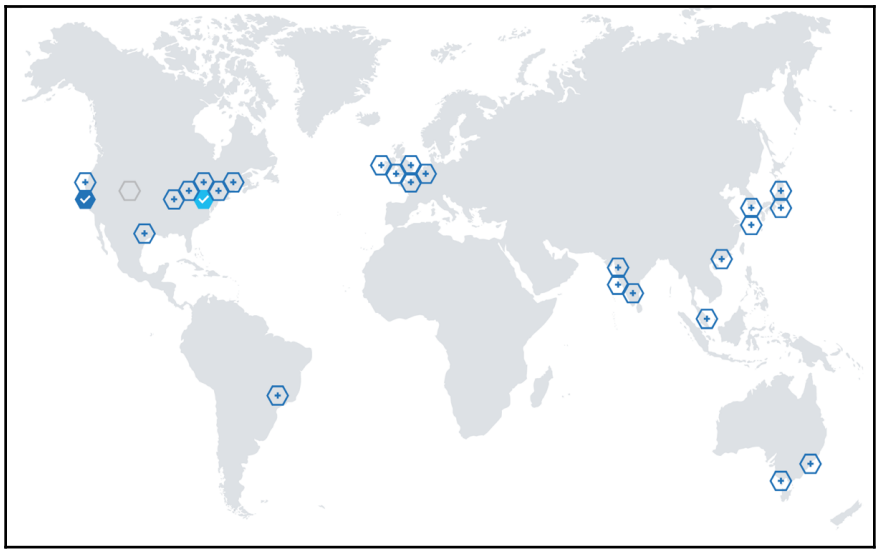
Let's perform a bottom-to-top analysis to have a better understanding of this database service. The following are three main features that Cosmos DB provides that establish pillars for supporting additional features:

- Partitioning
- Replication
- Resource governance

Partitioning makes it possible for Cosmos DB to provide an elastic scale out of storage and throughput by distributing the data in multiple logical and underlying physical partitions. We can start with something very small and grow elastically and seamlessly to something very large, increasing both storage and throughput as required. For example, we can start with a total storage size measured in gigabytes and end up with petabytes. We can start with small throughput requirements per second and end up with huge throughput requirements per second.

Replication makes it possible to deliver turnkey global distribution and replicate data through any number of regions in which Cosmos DB is available. The number of regions is continuously increasing and there are no limitations on the number of regions to which we can replicate data. For example, we can have a Cosmos DB database service working with the West US, East US, Brazil South, Japan East, and Japan West regions. The following diagram shows icons with sample regions in which a Cosmos DB database can be replicated (at the time of writing this book).

The hexagons represent the regions in which a database can be replicated:



Cosmos DB offers five consistency models to enable us to select the most appropriate one based on the most convenient write performance and the desired consistency. This way, we can manage performance with respect to consistency. We will analyze them in detail later in this chapter.

Resource governance makes it possible to provide high availability. Cosmos DB can provide 99.99% (also known as four nines) of availability in a single region and 99.999% (also known as five nines) of availability in multiple regions. Availability is one of the most important aspects of a database. Cosmos DB provides high availability in a transparent and automatic way that doesn't require manual changes in the configuration; that is, we don't need to make changes or redeploy and we can continue using the same endpoint.

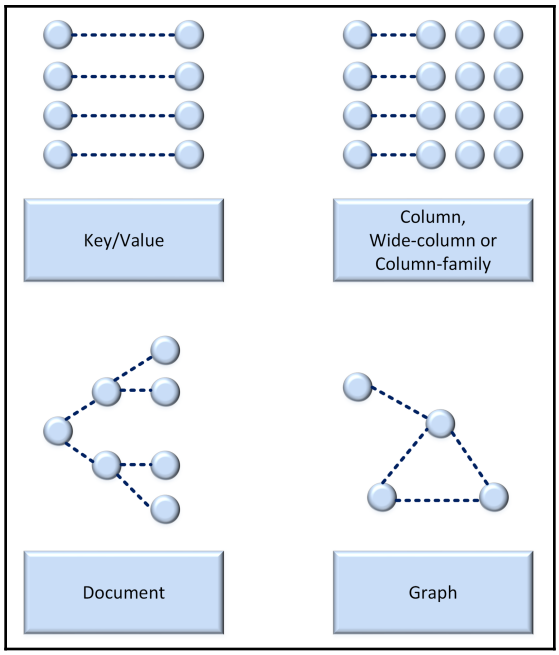
Of course, one of the key aspects of a database service is performance. Cosmos DB provides the necessary features for achieving predictable performance. The database service implements resource governance at a very fine level of granularity and on a per-request basis. This way, the database service guarantees a pre-configured desired throughput as well as the latency for each individual request. Hence, capacity planning is really straightforward.

Understanding the supported NoSQL data models

There are many flavors of NoSQL database. The following are the four most common types of NoSQL database:

- **Key/value**: This is a persistent dictionary. It is best for when we know the key and we need to retrieve the associated value for the key.
- **Column, wide-column, or column-family**: This organizes related data into columns instead of the typical organization in rows. It is best for when we need to query across specific columns in the database.
- **Document**: This allows persisting JSON objects (documents), which can include nested objects or arrays of other objects.
- **Graph**: This allows you to persist edges and nodes with their properties. It is best for when we need to store and navigate through complex relationships.

The following diagram outlines each of the four explained flavors of NoSQL database to make it easy to understand the typical data they persist:



Cosmos DB uses a schema-agnostic data store on top of the previously explained main features that provide a core platform. Cosmos DB can efficiently project this data store to the four previously listed NoSQL data models. Thus, the database service allows us to select the most appropriate NoSQL data model based on our needs, and we can take full advantage of partitioning, replication, and resource governance with any of them.

Using the appropriate API for each data model

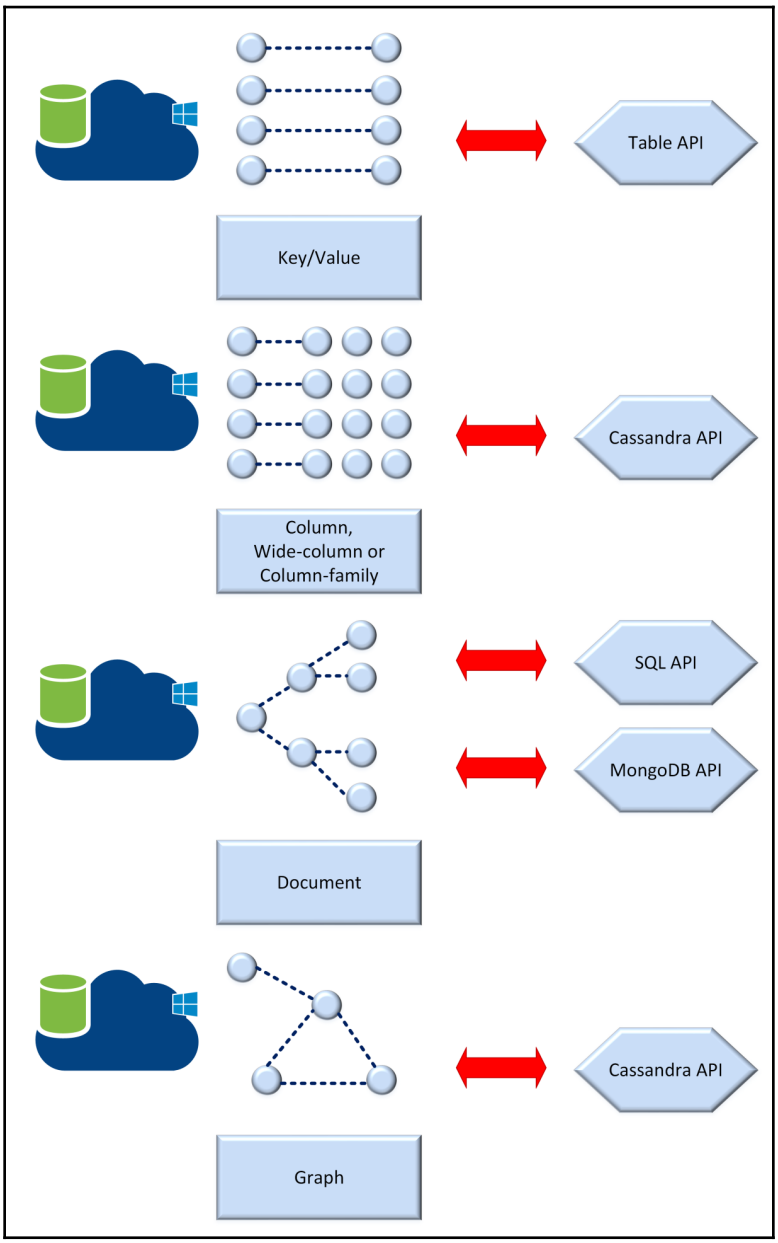
Under the hood, Cosmos DB internally stores data in a format called **Atom-Record-Sequence** (**ARS**), which is highly optimized for partitioning and replication. Hence, no matter the NoSQL data type and API, the data ends up stored in this internal format.

Cosmos DB provides support for five different APIs with SDKs for many programming languages and platforms. Based on the data model we use with our database, we must use a specific API to interact with the Cosmos DB database service. The following table summarizes the five APIs that are available based on the four data models:

NoSQL database type	**Available APIs**
Key/value	Table API
Column, wide-column, or column-family	Cassandra API
Document	SQL API MongoDB API
Graph	Gremlin API

Based on the information provided in the previous table, if we work with a document database, we can work with either the SQL API or the MongoDB API. If we are migrating an existing application that works with MongoDB to Cosmos DB, we can take advantage of the use of the MongoDB API to migrate the application to the new database service. If we are building an application from scratch, we might consider the use of the SQL API, which provides a Cosmos DB dialect of SQL to work against a document database. We will cover both scenarios in this book. We will work with the SQL API with .NET and C#, and we will work with the MongoDB API with Node.js.

The following diagram shows graphics that represent each of the four explained flavors of NoSQL database and the APIs that can be used for each of them:



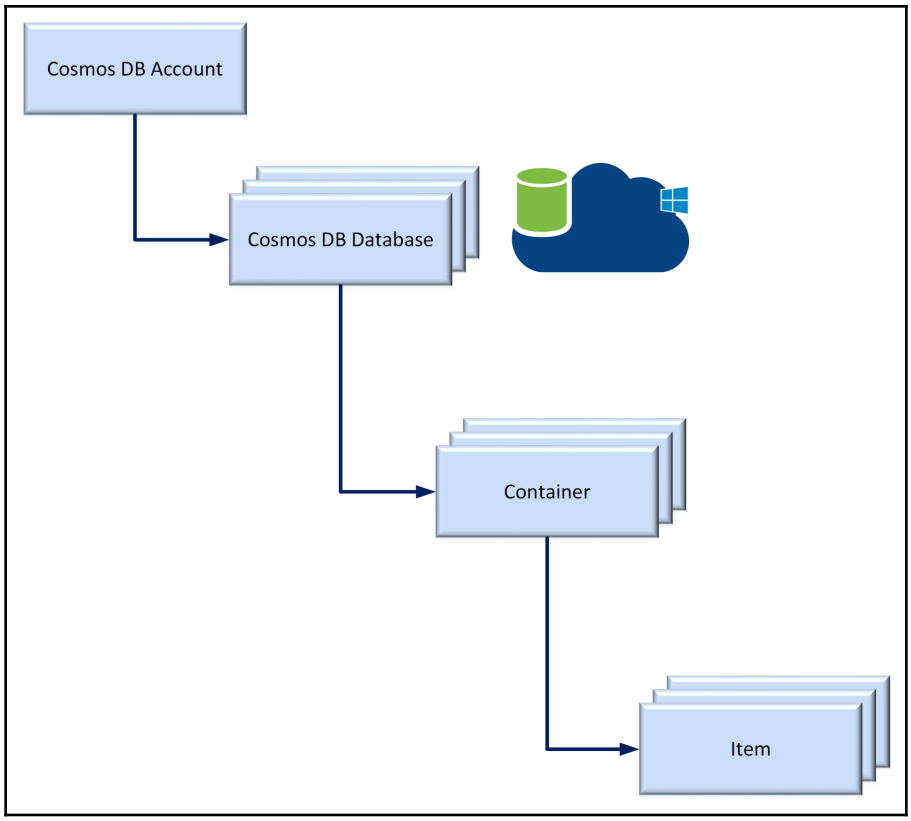
Diving deep into the Cosmos DB resource model

First, we must understand the Cosmos DB resource model, which is used by all supported NoSQL data models and some APIs. When we provision a new Cosmos DB account, we will be provided with a URI and an endpoint that represents the account and allows clients to establish a connection. At the time we provision the account, we must select the API that we want to use, and this selection will determine the type of NoSQL database that we will be creating, among other things, which we will learn about later. The following list shows the available APIs with the names used in the Azure portal and the type of NoSQL database that each of them will end up creating:

- **SQL**: Document
- **MongoDB**: Document
- **Cassandra**: Wide-column
- **Azure Table**: Key/value
- **Gremlin (graph)**: Graph

Once we have an account provisioned, we can create a new database that will use the API that was selected for the account. An account can have many databases of the same NoSQL type that use the same API.

The following diagram shows the generalized hierarchy of elements that belong to a Cosmos DB account:



Each database will have a set of containers whose name will be different based on the NoSQL database type and API. In fact, based on the NoSQL database type, the containers will be projected in a different way to the underlying data storage. The following list specifies the container name for each NoSQL database type:

- **Document**: Collection
- **Graph**: Graph
- **Key/value**: Table
- **Wide-column**: Table

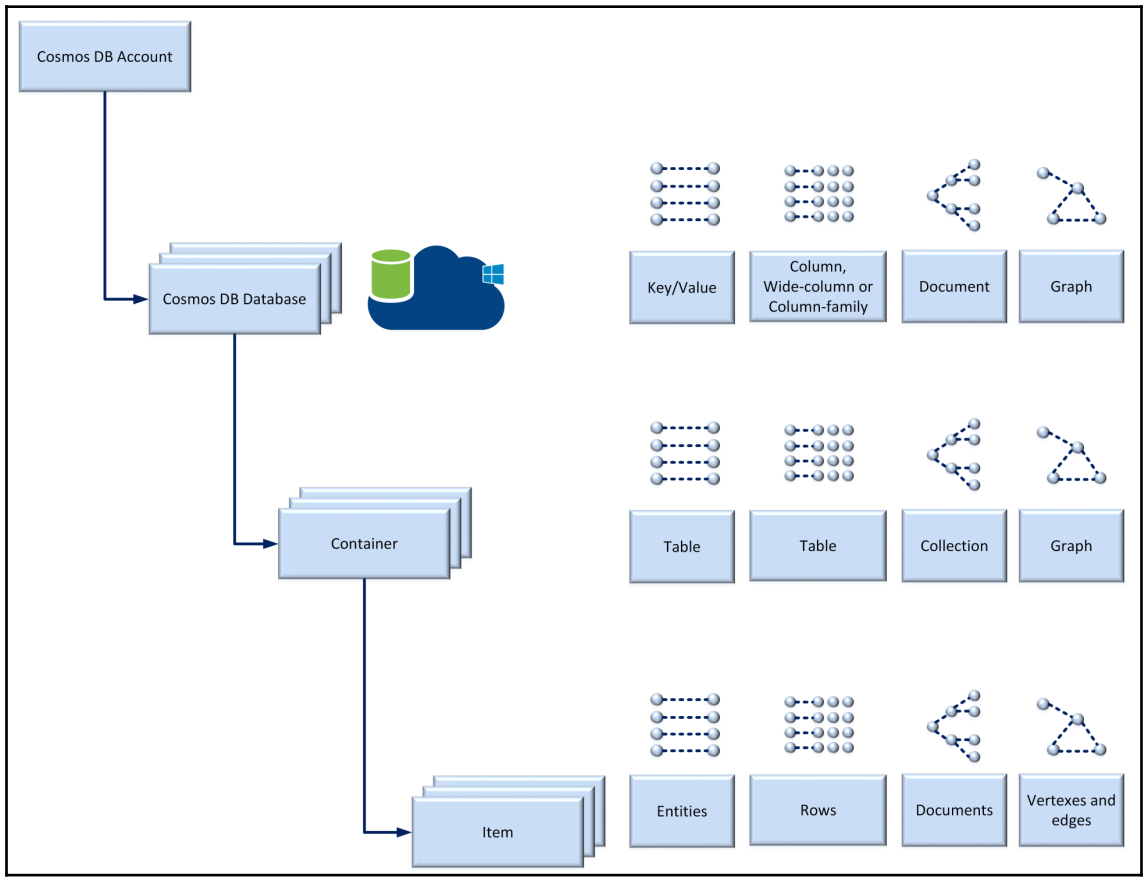
For example, when we work with a document database with either the SQL API or the MongoDB API, we will organize documents into containers known as **collections**. Whenever we create a new collection, we are able to provision the desired throughput, which we can then scale up or down on demand. We will also be able to specify a hint for how we want to distribute the data on the underlying partition sets. We will analyze each of these topics in detail later, as we want to stay focused on the Cosmos DB resource model for now.

Once we have a container provisioned, we can start storing data on it. One of the latest enhancements added in 2018 for this database service was the introduction of a multi-master capability. When we enable this feature, Cosmos DB allows us to write to our Cosmos DB containers in multiple regions at the same time with a latency of less than 10 milliseconds at the 99th percentile when we consume the Cosmos DB service within the Azure network. The multi-master feature makes it possible to use the provisioned throughput for databases and containers in all the available regions.

Each container will have a set of items whose names will be different based on the NoSQL database type and API. As is the case with the containers, based on the NoSQL database type, the items will be projected in a different way to the underlying data storage. The following list specifies the item name for each NoSQL database type:

- **Document**: Documents
- **Graph**: Vertexes and edges
- **Key/value**: Entities
- **Wide-column**: Rows

The following diagram shows the generalized hierarchy of elements that belong to a Cosmos DB account with the appropriate names based on the NoSQL database type on the right-hand side:



In addition, there are other container-level resources for server-side programmability that enable multi-record transactions within the partition key. We can write these resources in ECMAScript 2015 JavaScript:

- Stored procedures
- Triggers
- User-defined functions, also known as UDF

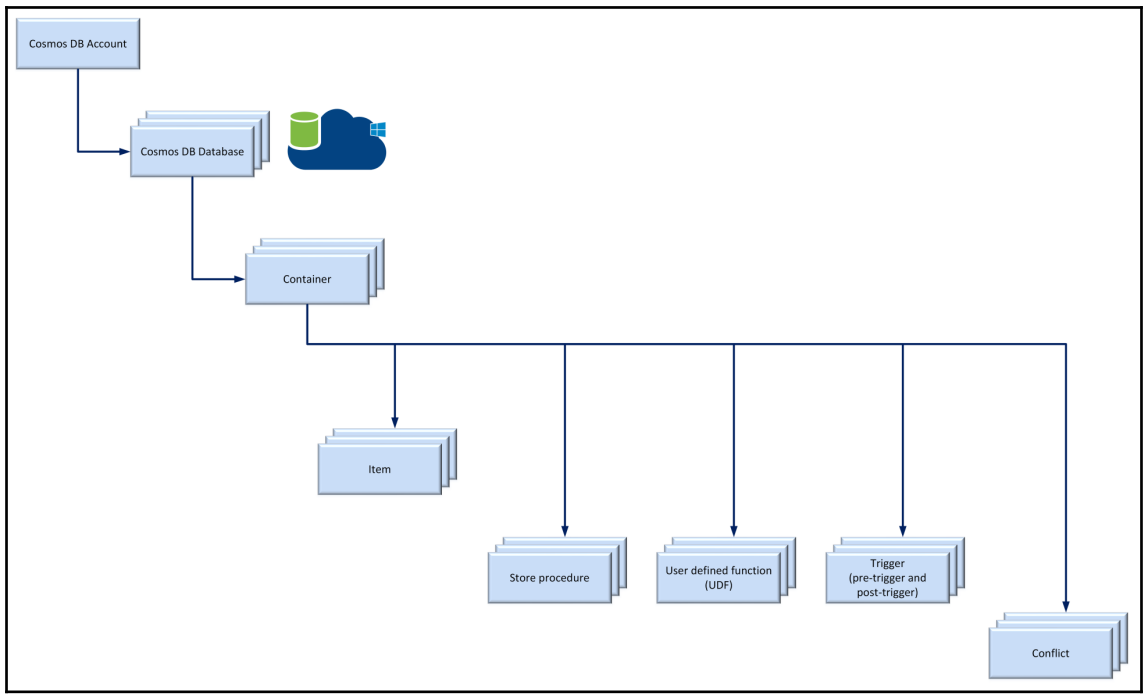
When we work with document databases, stored procedures allow us to operate on any document in the collection in which the stored procedure is defined.

We can write triggers that will be executed when specific operations are performed on a document. We can define pre-triggers, which are executed before the operation is performed; and post-triggers, which are executed after the operation is performed.

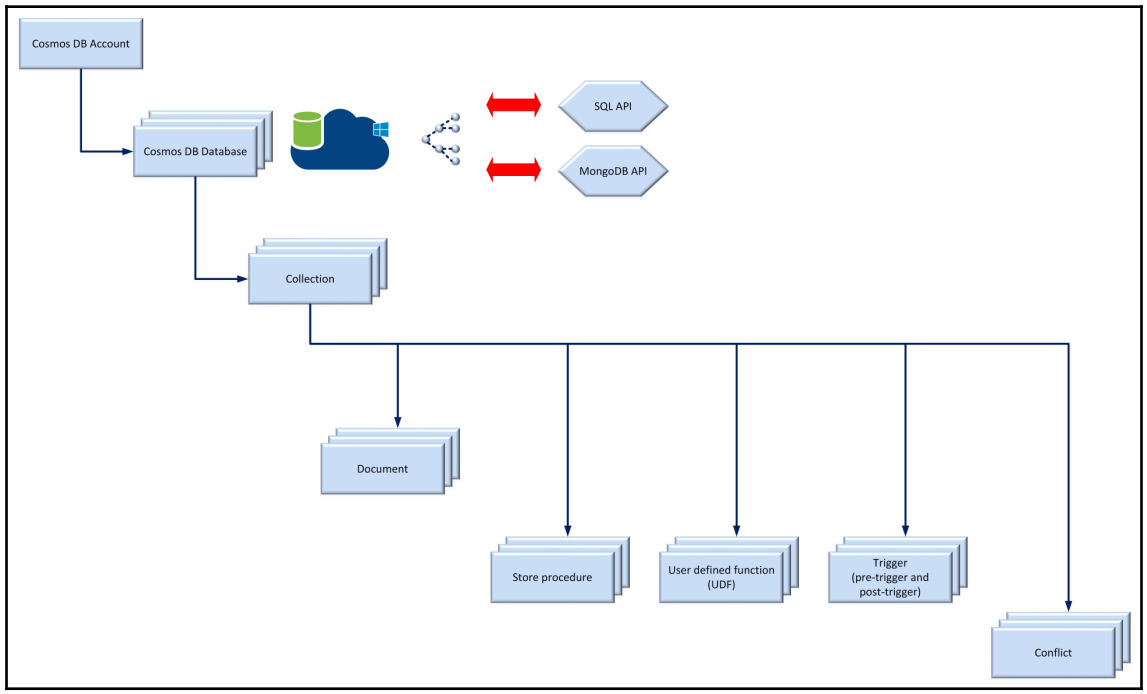
We can declare user-defined functions to extend the Cosmos DB query language's grammar and provide functions that implement custom business logic.

If a version conflict occurs on a resource for any operation, the conflicting resource will be persisted in a conflict feed within the container.

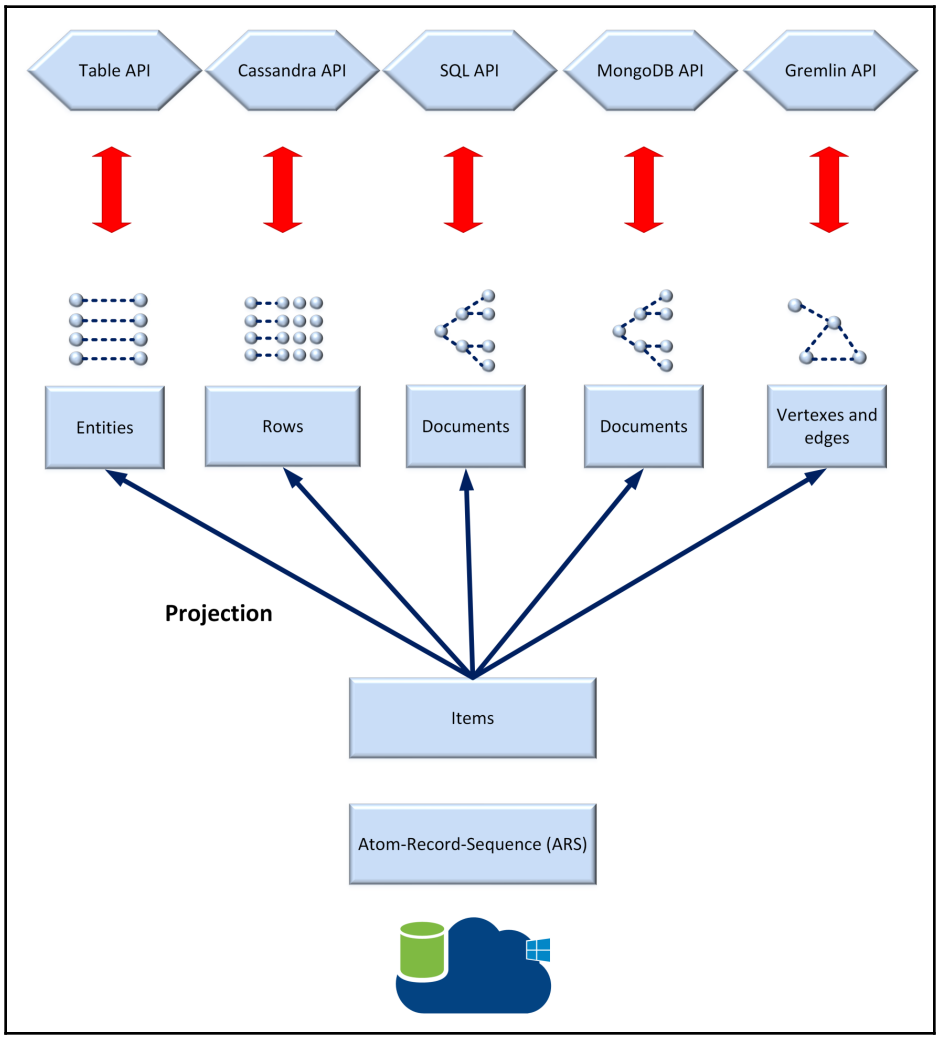
The following diagram shows the generalized container-level resources that belong to any Cosmos DB container:



The following diagram shows the generalized collection-level resources that belong to a Cosmos DB collection for a document database that uses either the SQL API or the MongoDB API:



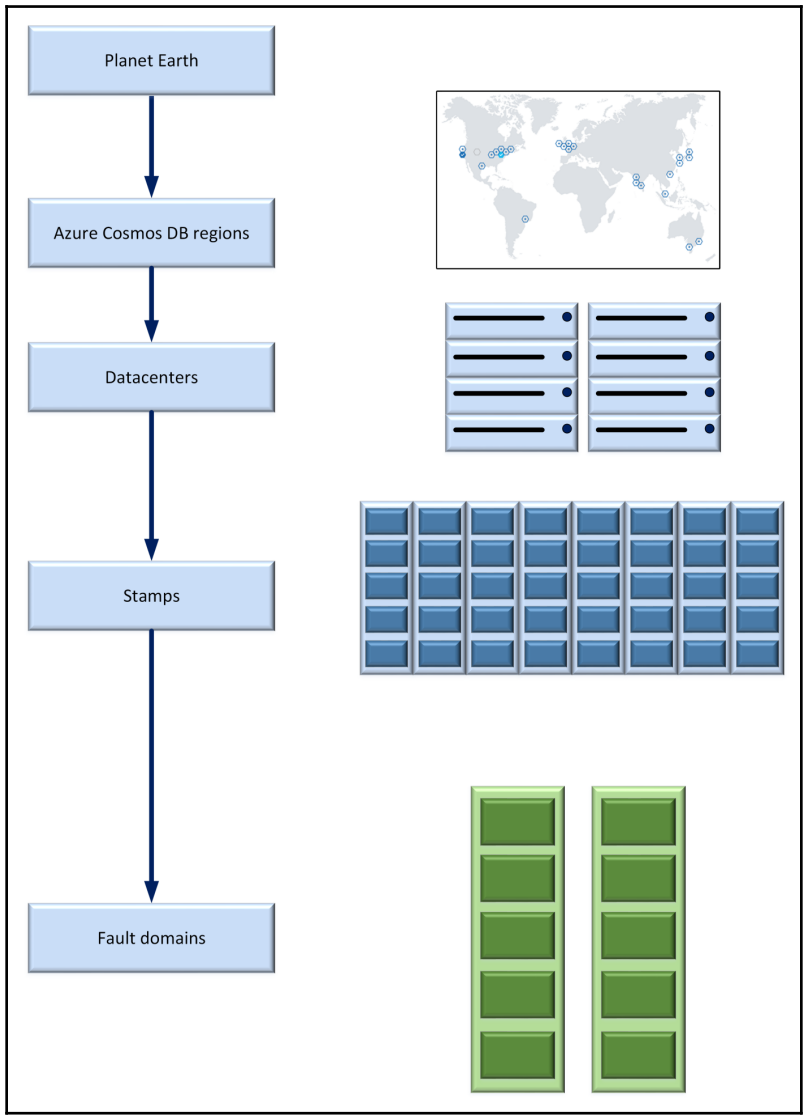
The following diagram illustrates the way Cosmos DB projects the data stored in the ARS format to the appropriate individual item for the different supported NoSQL database types and APIs:



It is very important to understand the Cosmos DB resource model and the name used to identify each element, because we will be working with its different components throughout this book, as well as the different examples.

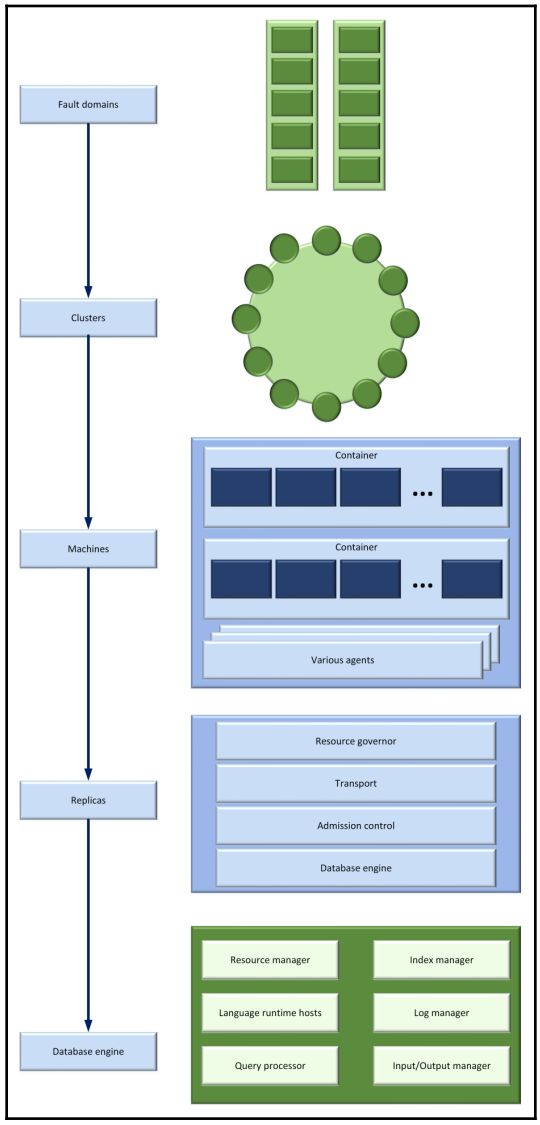
Understanding the system topology NoSQL

Now that we understand the basics of the Cosmos DB resource model, we will analyze the system topology that is hidden behind the scenes and makes it possible to run the database service at a global scale. The following diagram illustrates the system topology, starting at a Cosmos DB account on Earth, covering up to the fault domains. At the time I was writing this book, Azure didn't have any Moon or Mars regions enabled for Cosmos DB:



As previously explained, Cosmos DB is available in many Azure regions across around the world. Each Azure region has many data centers. Each data center has deployed many big racks known as **stamps**. The stamps are divided into fault domains that have server infrastructures.

The following diagram illustrates the system topology for each fault domain:

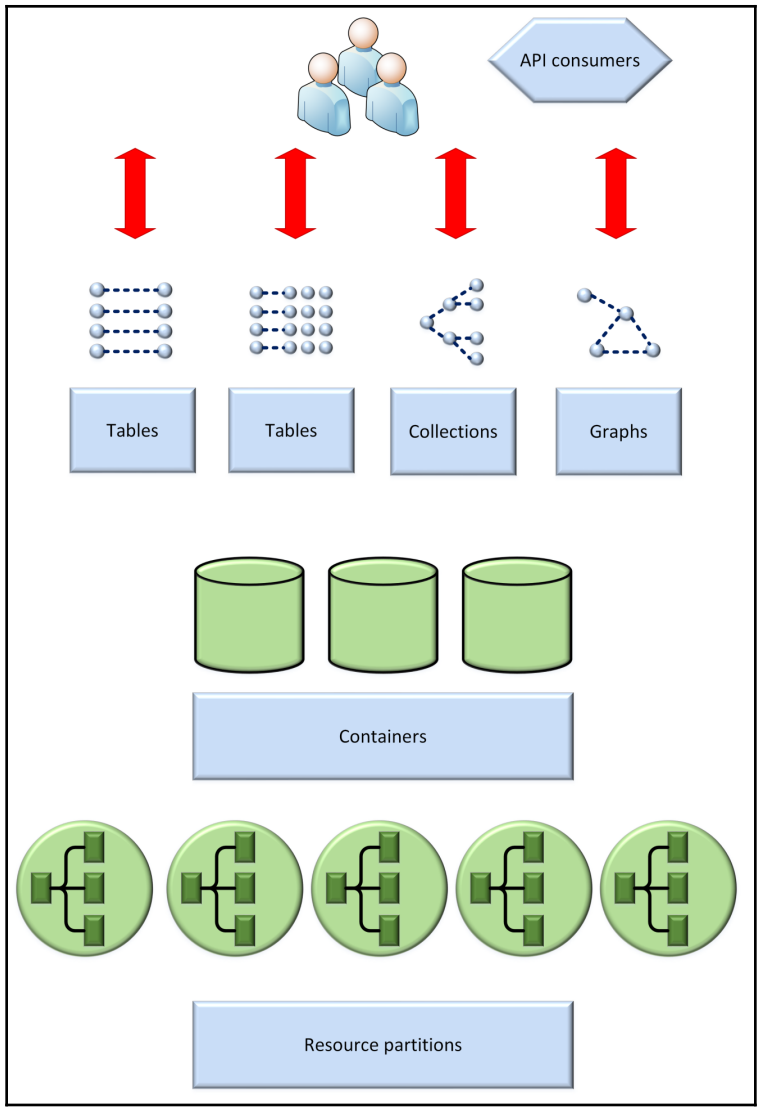


There are clusters with hundreds of servers deployed to many fault domains. The replica sets are deployed to many fault domains to provide an infrastructure that is highly resilient and continues working without issues when hardware failures occur. Each cluster has a database replica with the following elements:

- Resource governor for throughput and latency guarantees
- Transport layer for replication
- Admission control for security (authentication and authorization)
- Database engine to run operations, queries, and indexing

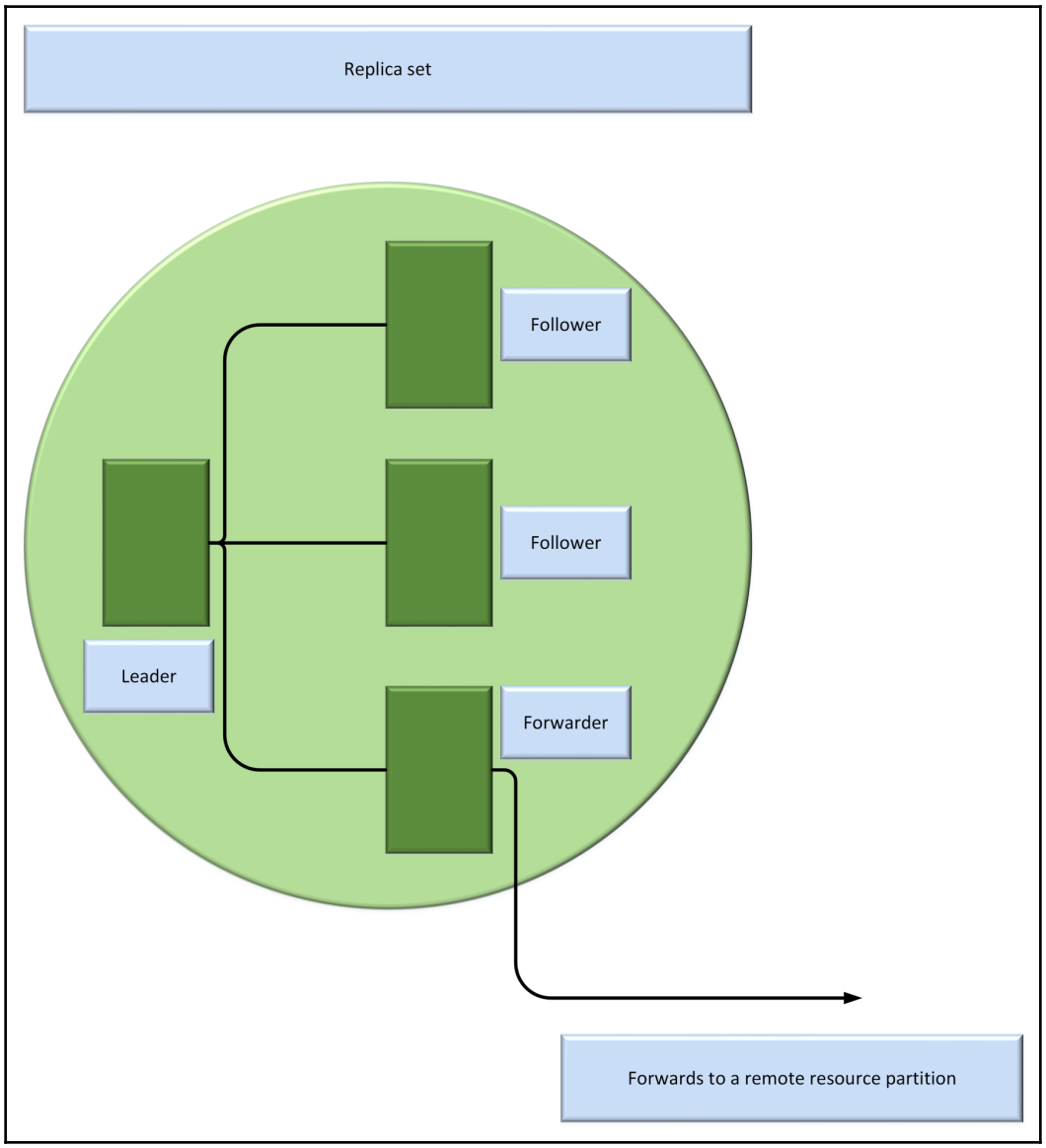
Learning about the resource hierarchy for each container

The following diagram shows the resource hierarchy for each container. For example, as previously learned, in a document database, the container is a collection:



The containers are the logical resources that are exposed to the APIs as collections, graphs, or tables. Each container has partition sets, which are composed of database replicas. The database service hosts four replicas per region. This way, whenever there are either hardware or software updates, they are completely transparent to us and we can continue working with the remaining replicas.

Resource partitions provide resource-governed coordination primitives. The following diagram shows a replica set in detail. Notice that each replica set hosts an instance of the database engine:



The database service is always online and available. The software and hardware updates on the Azure side happen under the hood for one out of four replicas per region while the remaining replicas continue working. Hence, we don't have to worry about availability due to operating system or database engine updates.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. In a single region, Cosmos DB can provide:
    1. 99.99% (also known as four nines) of availability
    2. 99.999% (also known as five nines) of availability
    3. 99.9999% (also known as six nines) of availability
2. In multiple regions, Cosmos DB can provide:
    1. 99.99% (also known as four nines) of availability
    2. 99.999% (also known as five nines) of availability
    3. 99.9999% (also known as six nines) of availability
3. Which of the following database type supported by Cosmos DB can work with either the SQL API or the MongoDB API:
    1. Key/value
    2. Column-family
    3. Document
4. The name for the container in a document database in Cosmos DB is:
    1. Key/value
    2. Document container
    3. Collection
5. The name for the item in a document database in Cosmos DB is:
    1. Value
    2. Document
    3. Row

Summary

In this chapter, we learned about the three main features of Cosmos DB that establish pillars for supporting additional features: partitioning, replication, and resource governance. We covered the four NoSQL data models supported by Cosmos DB and saw how they relate to the five available APIs.

Then, we learned about the different elements of the Cosmos DB resource model, allowing us to have a clear understanding of how to work with this database service. We understood the system topology that provides support to Cosmos DB at a global scale and we analyzed the resource hierarchy for each container. We now know the name for each element that we will have to use to develop applications that work with Cosmos DB and to manage this innovative database service.

Now that we understand the basics of Cosmos DB, we will provision a Cosmos DB account with the SQL API and we will start working with a document database, its collections and documents, which are the topics we are going to discuss in the next chapter.

2
Getting Started with Cosmos DB Development and NoSQL Document Databases

n this chapter, we will provision a Cosmos DB account that uses the SQL API and we will start working with a document database, its collections, and documents. We will work with Cosmos DB on Azure and we will also learn how to work with the Azure Cosmos DB Emulator. We will work with different tools to interact with our Cosmos DB document database that will be extremely useful for our common development tasks. We will do the following:

- Understand URIs, read-write and read-only keys, and connection strings
- Create a new document database with the SQL API
- Create a new collection
- Populate a collection with documents
- Understand automatically generated key-value pairs
- Understand schema-agnostic features
- Work with the web-based Azure Cosmos DB Explorer
- Use Azure Storage Explorer to interact with Cosmos DB databases
- Work with the Azure Cosmos DB Emulator

Provisioning a Cosmos DB account that uses the SQL API

In `Chapter 1`, *Introduction to NoSQL in Cosmos DB*, we learned about the different elements of the Cosmos DB resource model. Now we will use the web-based Azure portal to provision a Cosmos DB account with the SQL API. Then, we will create a document database, generate a collection for this database, and populate the collection with many JSON documents.

In order to follow the next steps, you have the following three options:

- Work with an already active Azure account
- Sign up for a new Azure account
- Use the limited-time try Cosmos DB for free feature

If you want to work with an already active Azure account, you have to take into account that you will spend credits and, based on your free resources or credits, you can end up being billed for the storage, request units, and bandwidth consumed.

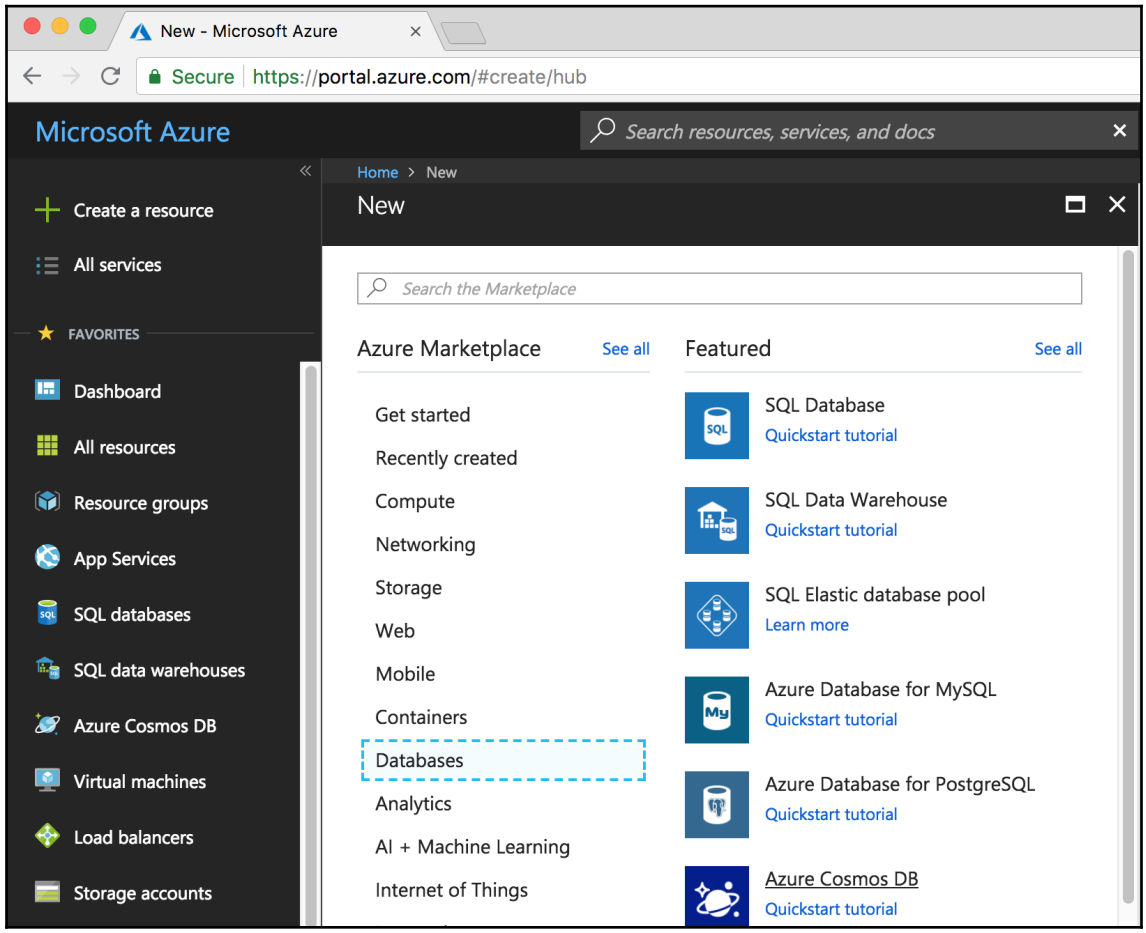
If you don't have an active Azure account, you can take advantage of the free $200 in Azure credits that you will receive to explore the services for 30 days when you sign up for a new Azure account. After 30 days, you will continue to have the account available to you, but you will start to be billed for the services. Note that the conditions usually change, so make sure you review them before signing up for these kinds of offers.

At the time I was writing this book, Microsoft made it possible to try Cosmos DB for free for a limited time, with no subscription or credit card number required. If you want to use this option to work with the samples in this book, you can sign up for a free account here: `https://azure.microsoft.com/en-us/try/cosmosdb/`.

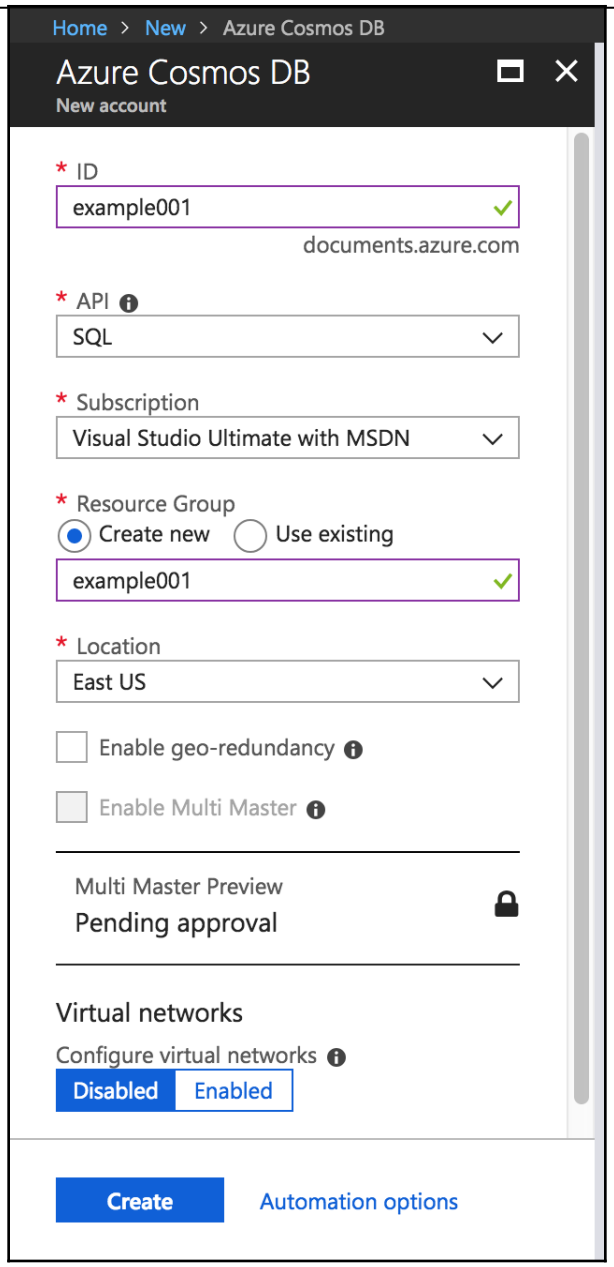
If none of the previous options are suitable for you, you can work with the Azure Cosmos DB Emulator. However, it is highly recommended that you take a look at how things work on the Azure portal before moving to the emulator. In most cases, it will be convenient to combine work with Cosmos DB in Azure with the use of the emulator to reduce software development costs. We will analyze the use of the Azure Cosmos DB Emulator later in this chapter, in the *Working with the Azure Cosmos DB Emulator* section. So, make sure you don't skip that section.

Once you have selected the most convenient option for you, open a web browser and sign in to Microsoft Azure in the Azure portal, `https://portal.azure.com/`:

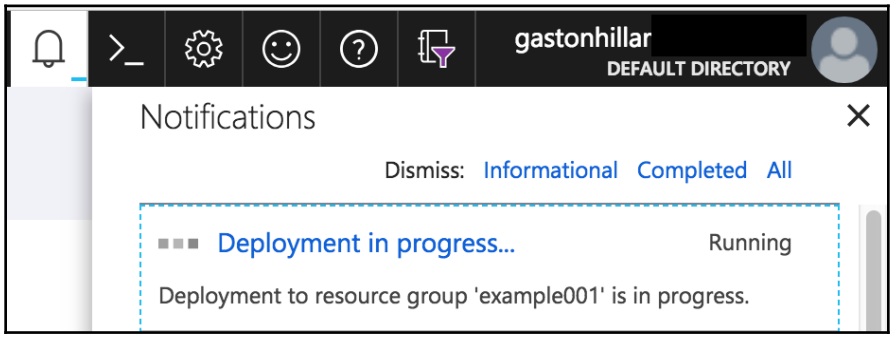
1. Click **Create a resource** | **Databases** | **Azure Cosmos DB**. The following screenshot shows these options in the Azure portal:



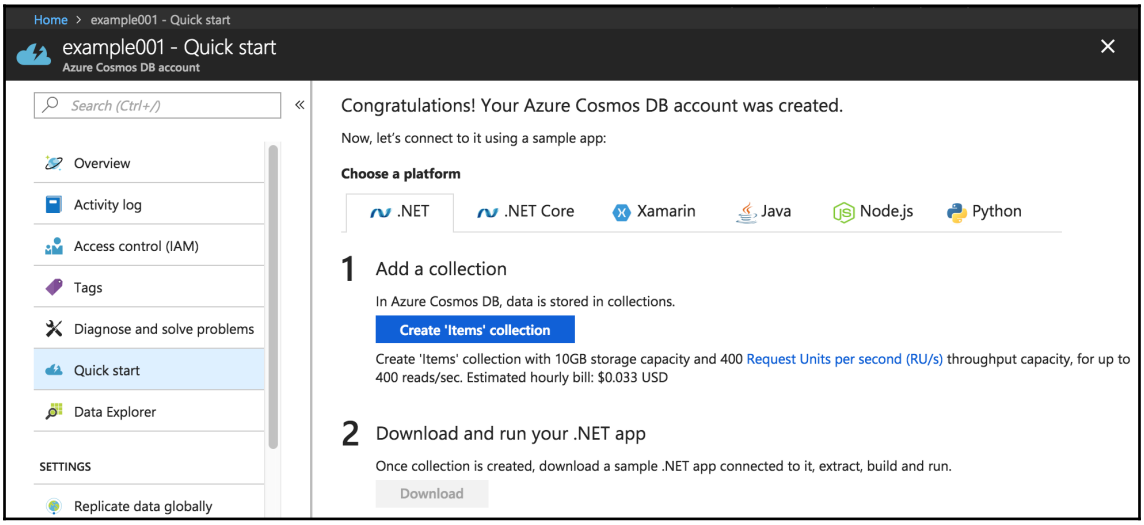
2. Enter a unique name to identify the new Azure Cosmos DB account in the ID textbox. Note that Azure will append `.documents.azure.com` to the value you enter for the ID textbox. The name can contain only lowercase letters, numbers, and the hyphen (-) character, and must be between 3 and 31 characters. In this case, I will use `example001`, Azure will generate an ID equal to `example001.documents.azure.com`, and the URI for the account will be `https://example001.documents.azure.com:443/`. Make sure you specify a value that the portal indicates as available after you enter it.
3. Select **SQL** in the **API** dropdown. This way, we will be able to create document databases that will use the SQL API in the account.
4. Select the Azure subscription that you want to use for the new Cosmos DB account in the **Subscription** dropdown. Don't forget that your Azure subscription might be charged, and therefore you should review the selected subscription name twice before creating the account.
5. Select the **Create new** option in **Resource Group** and enter the same name specified in the ID textbox. In this case, I will also use `example001`.
6. Select the most convenient region for you, based on your location, in the **Location** dropdown. This selection must take into account the available latency and the charges associated with the location. In this case, I will select `East US`. We won't take advantage of geo-redundancy or multi-master features. We want the basic features for a Cosmos DB account. The following screenshot shows the **Azure Cosmos DB New account** panel with sample values for the different options:



7. Click **Create**. Azure will validate the selected options. If all the options are valid, Azure will start the necessary processes to provision the new Cosmos DB account.
8. Click on the **Notifications** icon (a bell at the top-right corner) and you will see a **Deployment in progress...** notification indicating that the processes for creating the new Cosmos DB account are running. The following screenshot shows a sample of the notification:



In a few minutes, the portal will finish the deployment and it will display a **Quick start** panel indicating that the Cosmos DB account has been created. The following screenshot shows that panel, which provides quick start tabs for establishing a connection with the new account with diverse programming language and platform options available:

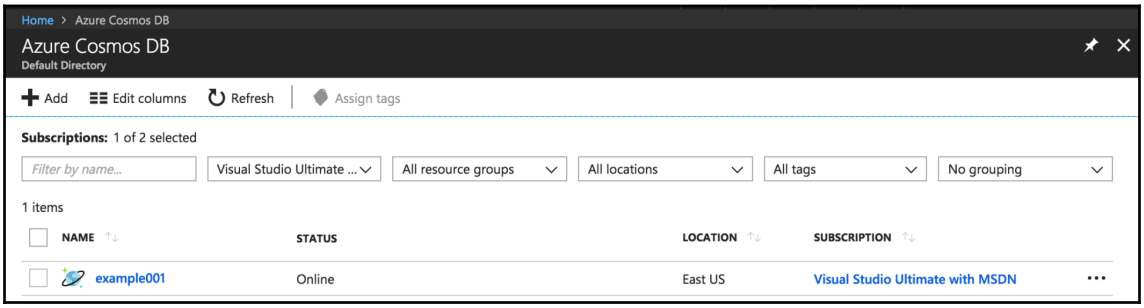


In this case, we follow the necessary steps to create a Cosmos DB account. As happens with most tasks in Azure, it is possible to automate these tasks that create resources with different scripting mechanisms, such as the cmdlets provided by Azure PowerShell and the Azure Resource Manager templates.

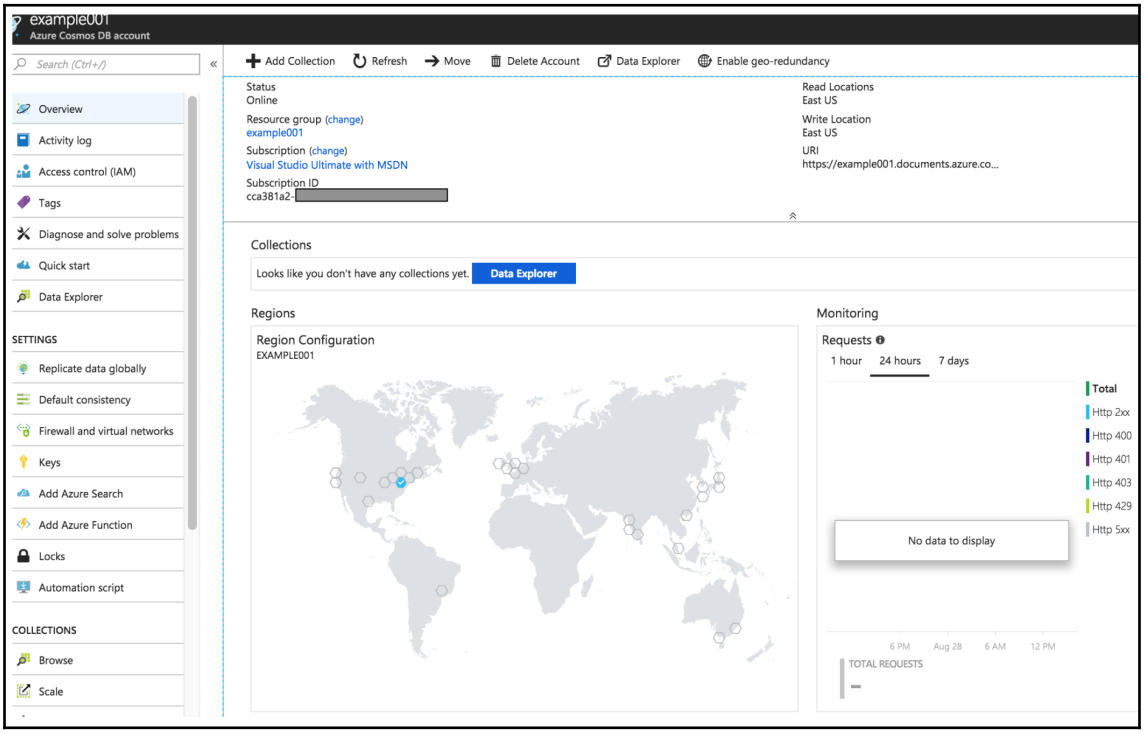
Understanding URIs, read-write and read-only keys, and connection strings

After you create a new Cosmos DB account, you will see the **Quick start** panel. However, the next time you sign in to the Azure portal, you won't see this panel. So, we will follow the necessary steps to access the recently created Cosmos DB account panel, considering we signed out and then signed in to the portal again:

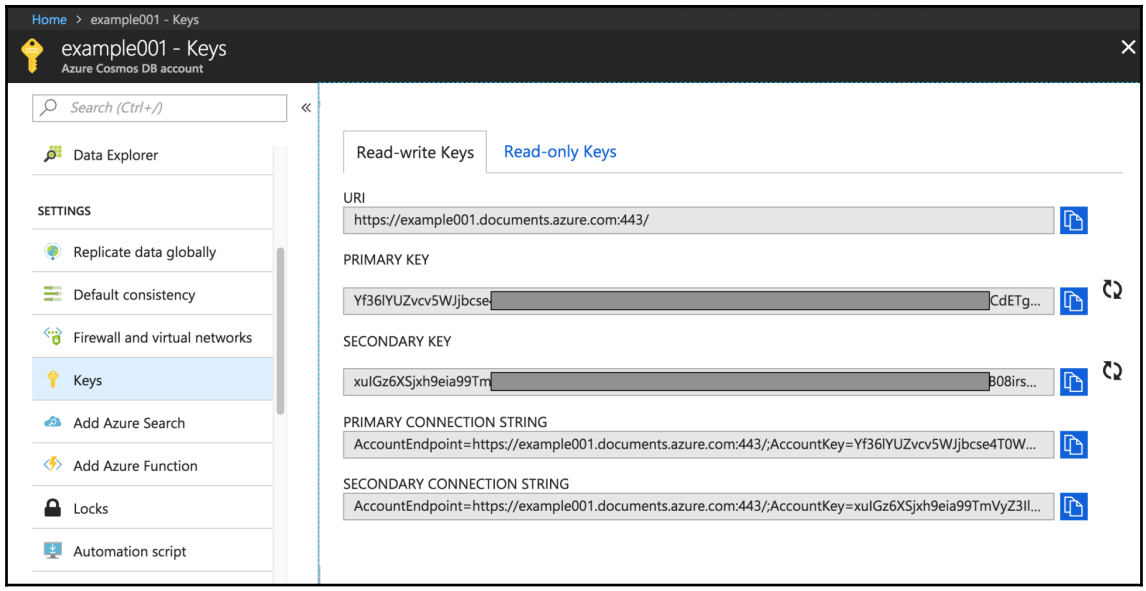
1. Click **Azure Cosmos DB** in the left-hand panel. The portal will display a list with all the Cosmos DB accounts, their statuses, locations, and the subscription to which they belong. Note that if you have multiple Azure subscriptions, the list will include only the accounts for the selected subscription. Make sure you select the subscription with which you have created the account. The following screenshot shows a sample list that includes the `example001` Cosmos DB account:



2. Click on the name for the recently created Cosmos DB account (`example001`) and the portal will open the page for this account with a menu at the left-hand side, and it will select the **Overview** option, which displays a panel at the right-hand side. **Overview** shows a toolbar at the top, information about the account in the top panel, and the **Collections**, **Regions**, and **Monitoring** panels. This page will be our entry point for performing many operations for our new Cosmos DB account in the Azure portal. The following screenshot shows some of the elements of the page and the **Overview** panel. Note that, in this case, the Cosmos DB account has only one region configured:



3. Click on the **Keys** option on the left-hand side menu, located below the **SETTINGS** title. The portal will display the **Keys** panel with two tabs: **Read-write Keys** and **Read-only Keys**. The following screenshot shows sample information provided by the **Read-write Keys** tab:



The **Read-write Keys** tab provides the following data, which we can copy to use different tools, SDKs, and programming languages to establish a read-write connection with the Cosmos DB account:

- **URI**: This is also known as the endpoint URL or Cosmos DB account endpoint. The URI includes the ID value we used to create our Cosmos DB account. For example, if the ID value is `example001`, the URI is `https://example001.documents.azure.com:443/`.
- **Primary key**: We can use this key as the authorization key in combination with the URI to establish a read-write connection with our Cosmos DB account.
- **Secondary key**: If we don't want to share the primary key, we can provide the secondary key to be used as the authorization key in combination with the URI to establish a read-write connection with our Cosmos DB account. In addition, the availability of two keys makes it possible to perform key rotation without service disruption.
- **Primary connection string**: This is a connection string that uses the URI and the primary key as the authorization key.
- **Secondary connection string**: This is a connection string that uses the URI and the secondary key as the authorization key.

In some cases, we will use the URI and the primary key, and in other cases, we will use the primary connection string to establish a read-write connection with our Cosmos DB account. It is possible to regenerate both the primary and secondary keys by clicking on the regenerate icons located at the right-hand side of the copy icon of each key. If we regenerate a key, the related connection string will be updated to use the new key.

Whenever we require the URI, primary key, or primary connection string in the next sections, we will be using the values retrieved in the **Read-write keys** tab.

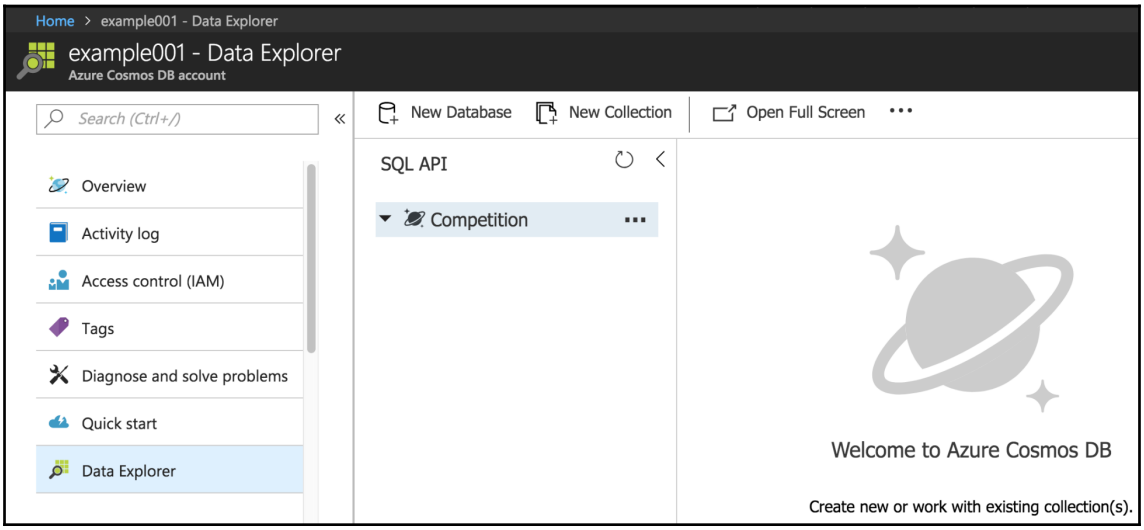
If you need to work with read-only keys because you don't want to enable changes in the Cosmos DB account for any specific connection, you can use the keys and connection strings provided in the **Read-only Keys** tab.

Creating a new document database with the SQL API

We have a Cosmos DB account that works with the SQL API but we still don't have a database. Now we will continue working with the Azure web-based portal and we will create a new document database in the previously created account. We must take into account that we can use other tools, which we will analyze later, to perform this task. It is also possible to automate the task through scripting and to create a new database by using one of the available SDKs. However, it is a good idea to learn how things work in the portal first:

1. Make sure you are on the page for the Cosmos DB account in the portal. Click on the **Data Explorer** option on the left-hand side menu. The portal will display the **Data Explorer** panel, which will allow us to work with the databases for the current Cosmos DB account. In this case, the panel won't list any databases because we haven't created any yet.
2. Click on **New Database** in the toolbar located at the top of the panel. The **New Database** panel will appear.

3. Enter `Competition` in **Database id**. We don't want to provision throughput at the database level, so leave the **Provision throughput** checkbox unselected. This way, we will provision throughput per collection.
4. Click **OK**. Azure will create the new document database with the SQL API and, after a few seconds, the database name will be added to the list of databases at the left-hand side, below the SQL API title, as shown in the next screenshot:

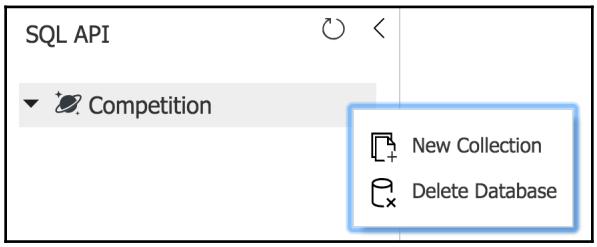


After you finish working with the examples, make sure you delete all the resources you have been creating when you don't need them anymore. If you don't delete the resources and you work with an Azure subscription instead of using the emulator, you will spend any remaining Azure credits and you could end up being billed.

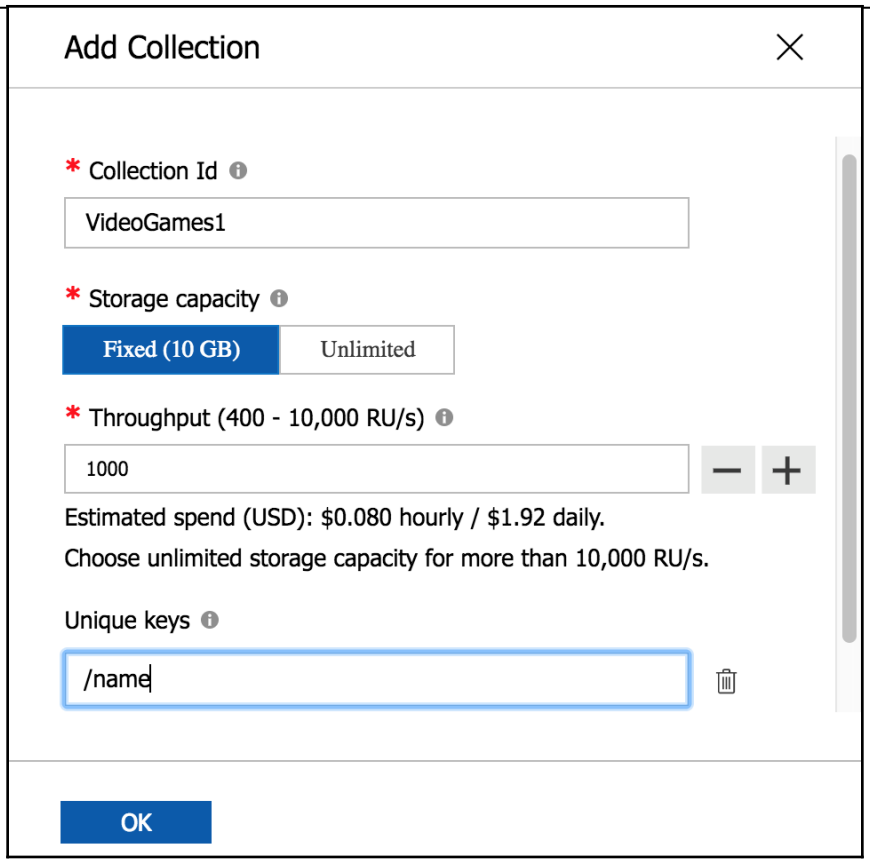
Creating a new collection

Now we will create a new collection for the database.

1. Hover on the database name (`Competition`) and click on the ellipses at the right-hand side (...) to display a context menu, as shown in the next screenshot:

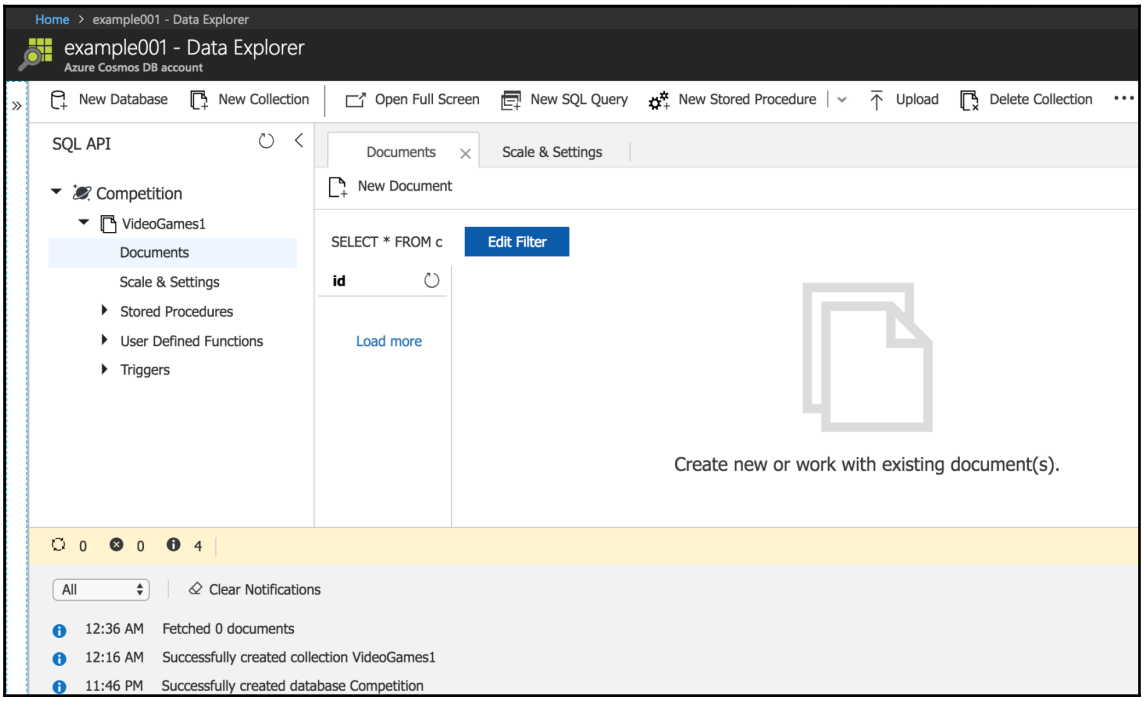


2. Click on **New Collection** in the context menu. The **Add Collection** panel will appear.
3. Enter `VideoGames1` in **Collection Id**.
4. Select **Fixed (10 GB)** in **Storage capacity**. This way, the maximum storage size for the collection will be 10 GB. In this case, we won't configure a partition key.
5. Enter `1000` in **Throughput (400 - 10,000 RU/s)**. This way, Azure will provision 1,000 request units per second for this collection. Note that based on the value we enter, the portal displays the estimated spend below this field.
6. Click **+ Add unique key** and a new textbox will be added below **Unique keys**. Enter `/name` in this textbox to specify that we want the **name** value to be unique per partition key. In this case, there is only one partition, and therefore, the data integrity layer will make sure the **name** value will be unique in this collection. The following screenshot shows the **Add Collection** panel with the values explained:



7. Click **OK**. Azure will create the new document collection named `VideoGames1`.
8. Click on the database name (`Competition`) at the left-hand side, below the **SQL API** title, and the new collection will be shown. Click on the `VideoGames1` collection and you will see the following items:
    - **Documents**
    - **Scale & Settings**
    - **Stored Procedures**
    - **User Defined Functions**
    - **Triggers**

In addition, the **Documents** tab will display a query that retrieves all the existing documents for the new collection. As we just created the collection, it is empty and the result set shows no documents. The following screenshot shows the **Documents** tab for the new collection and the notifications panel at the bottom, with the results of all the tasks we have been performing in the portal:



Populating a collection with documents

Now we will populate the recently created collection with a few documents that represent games that were part of eSports competitions with specific tags, definitions for their levels, and the high score achievements. We will continue working with the Azure web-based portal, but we will learn to use other tools to perform the same task in the following sections.

Create a new JSON file with the following contents in your favorite text editor and save it in a temporary folder as `videogame1_01.json`. The following lines show the code that defines the game with its tags, levels, and high score achievements. The code file for the sample is included in the `learning_cosmos_db_02_01` folder in the `videogames/videogame1_01.json` file:

```
{
    "id": "1",
    "name": "Battle Royale Kingdoms",
    "lastCompetitionDate": "2018-09-29T04:36:22.7251173Z",
    "tags": [
        "mobile", "2D", "card game"
    ],
    "levels": [
        {
            "title": "Training Camp for Dummies",
            "towers": 2,
            "towerPower": 30
        },
        {
            "title": "Jungle Arena",
            "towers": 2,
            "towerPower": 40
        },
        {
            "title": "Legendary World",
            "towers": 5,
            "towerPower": 100
        }
    ],
    "highestScores": [
        {
            "player":
            {
                "nickName": "Brandon in Wonderland",
                "clan": "Wonderland Warriors"
            },
            "score": "750"
        }
    ]
}
```

The JSON document has arrays and nested documents. The JSON document defines the following keys:

- `id`: The game ID. Note that we must use a string ID for the JSON documents that we want to store in a Cosmos DB SQL API collection.
- `name`: The game's name.
- `lastCompetitionDate`: The last date and time at which the game was played in an eSports competition.
- `tags`: An array of strings with tags related to the game.
- `levels`: An array of JSON objects that defines the different levels that the game has. In this case, each level has a title, a number of towers, and the tower's power value.
- `highestScores`: An array of JSON documents that specifies the players nicknames, clans, and their high scores for the game in eSports competitions. Note that the player is a nested JSON document with the `nickName` and `clan` keys.

Copy the content of the JSON document to the clipboard.

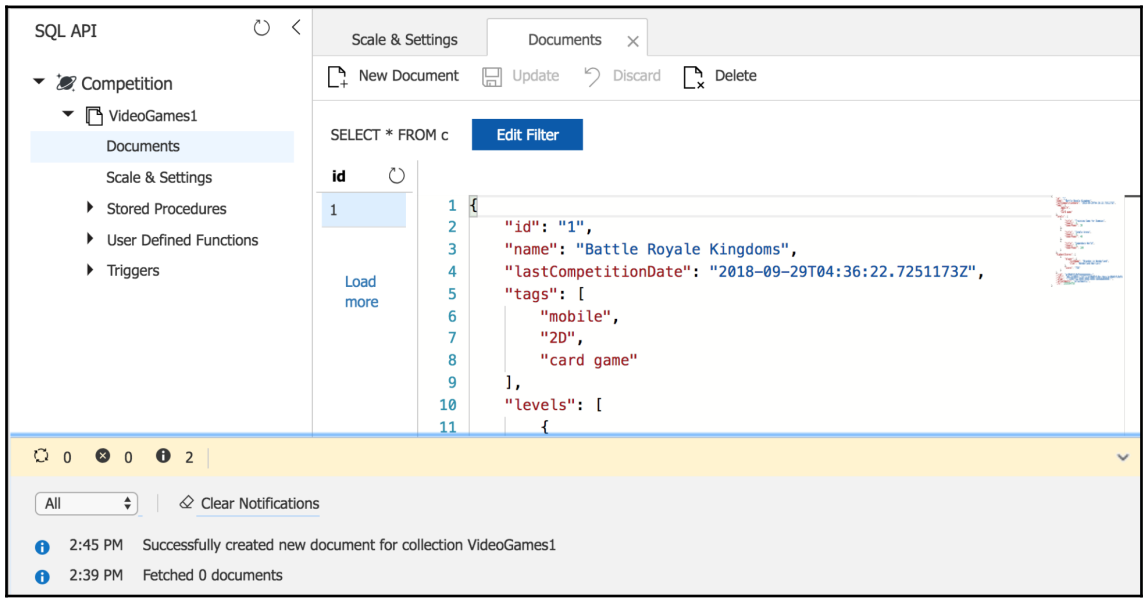
Click **New Document** on the toolbar below the **Documents** tab. The portal will display a JSON editor with the following lines that indicate that you must provide a string value for the `id` key:

```
{
    "id": "replace_with_new_document_id"
}
```

The `id` key is always required and represents the unique identifier that identifies the document within the collection. So, we won't be able to add another document with the same ID to the same collection. The ID must be a string with a maximum of 255 characters.

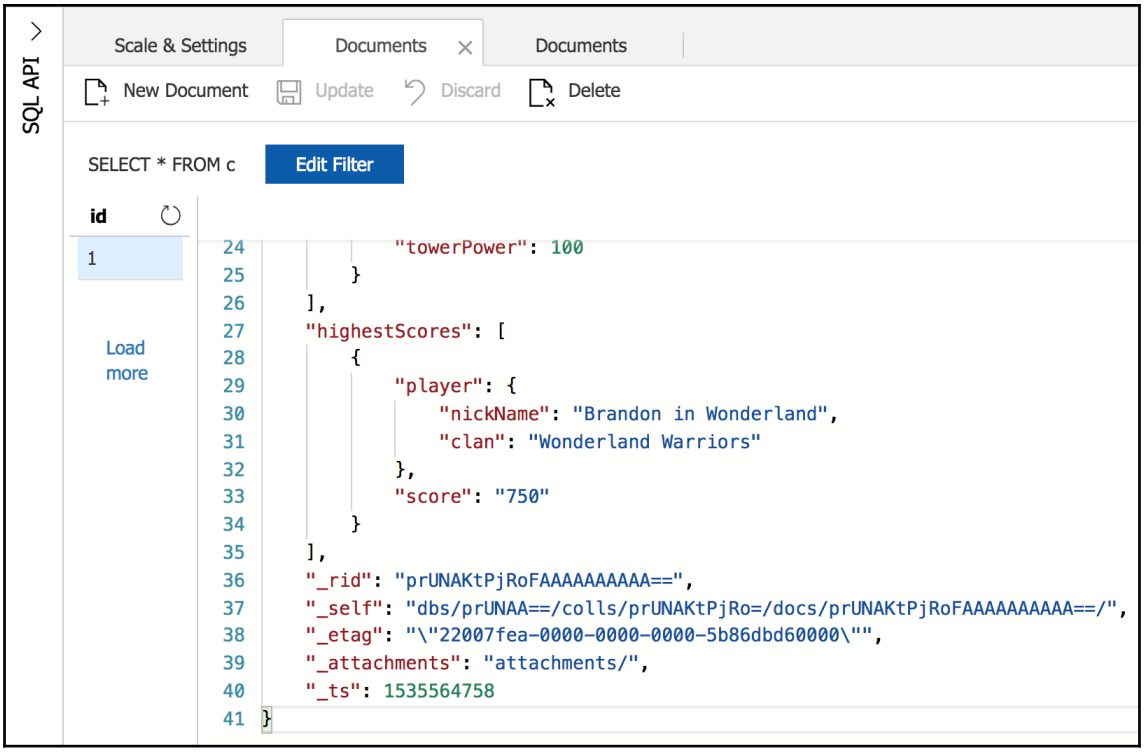
Select all the content in the editor and paste the content of the clipboard. This way, you will replace the previous lines with the JSON document you created with your favorite text editor for JSON content. The text editor provided by the portal uses syntax highlighting and error indicators for the JSON content. So, if the JSON document has issues, you will notice them before trying to insert a new document into the collection and the **Save** button will be disabled.

Click **Save** on the toolbar below the **Documents** tab. The new document will be inserted into the `VideoGames1` collection, and the portal will display the `id` for the new document in the list at the left-hand side and the inserted document in the editor at the right-hand side. The following screenshot shows the added document:



Understanding automatically generated key-value pairs

Scroll down to the latest lines of the document in the web-based editor and you will see new key-value pairs that were not part of the original document. The following screenshot shows the last lines in the embedded JSON editor:



The following lines show an example of the new key-value pairs. Note that many values will be different in your case:

```
    "_rid": "prUNAKtPjRoFAAAAAAAAAA==",
    "_self":
"dbs/prUNAA==/colls/prUNAKtPjRo=/docs/prUNAKtPjRoFAAAAAAAAAA==/",
    "_etag": "\"22007fea-0000-0000-0000-5b86dbd60000\"",
    "_attachments": "attachments/",
    "_ts": 1535564758
}
```

Cosmos DB added the following keys to our document, which start with an underscore (_) as a prefix and contain system-generated values for the document resource:

- `_rid`: This string defines the resource ID; that is, a unique identifier for the resource. Cosmos DB uses this resource ID internally to identify and navigate through the document resource.
- `_self`: This string provides a unique addressable URI for the resource, also known as a **self link**. When we work with the different SDKs, we can use the self link to easily identify a document or other resources that provide a self link. This URI is the combination of the self links for the database and the collection to which the document belongs. In fact, the self link includes the hierarchical paths to the resource stack. For example, the previous lines showed that the value for `_self` was `"dbs/prUNAA==/colls/prUNAKtPjRo=/docs/prUNAKtPjRoFAAAAAAAAAA==/"`, which includes the paths to the self links for the database resource, `"dbs/prUNAA=="` and the collection resource, `"dbs/prUNAA==/colls/prUNAKtPjRo="`. The self link uses a slash (`/`) as the path separator.
- `_etag`: This string stores the entity tag that Cosmos DB uses for optimistic concurrency control.
- `_attachments`: This string stores the addressable path of the attachment resource related to the document that provides information about the attachments. A document can contain many attachments, which are special documents that contain references and associated metadata with an external blob or media file.
- `_ts`: This integer stores a timestamp with the last date and time in which the document has been updated. Whenever a document is updated, the value for `_ts` will change.

In this case, we analyzed the system-generated key-value pairs for a document. These values are also added whenever we create other resources through an SDK, such as databases and collections. So, whenever we have to address a resource, we can take advantage of the self link; that is, the value of the `_self` property for the resource.

Understanding schema-agnostic features

Now we will add another document to the same collection. In this case, the new document will have new keys that didn't exist in the previously inserted document.

Create a new JSON file with the following contents in your favorite text editor and save it in a temporary folder as `videogame1_02.json`. The following lines show the code that defines a new game with its tags, levels, high score achievements, and prizes. The code file for the sample is included in the `learning_cosmos_db_02_01` folder in the `videogames/videogame1_02.json` file:

```
{
    "id": "2",
    "name": "Fortnite vs Zombies",
    "lastCompetitionDate": "2018-09-30T03:31:20.7251173Z",
    "tags": [
        "3D", "battle royale", "monsters", "shooter"
    ],
    "platforms": [
        "PS4", "XBox", "PC", "Switch", "iPad", "iPhone", "Android"
    ],
    "levels": [
        {
            "title": "Dancing in the storm",
            "maximumPlayers": 50,
            "minimumExperienceLevel": 30
        },
        {
            "title": "Rainbows after the storm",
            "maximumPlayers": 30,
            "minimumExperienceLevel": 60
        },
        {
            "title": "The last of us",
            "maximumPlayers": 10,
            "minimumExperienceLevel": 100
        }
    ],
```

```
        "highestScores": [
            {
                "player":
                {
                    "nickName": "PlaystationBoy",
                    "clan": "USA Players",
                    "experienceLevel": 140
                },
                "score": "5600"
            },
            {
                "player":
                {
                    "nickName": "KevinSwitchMan",
                    "clan": "Italian Warriors",
                    "experienceLevel": 125
                },
                "score": "3300"
            }
        ]
    }
```

The JSON document includes many keys and nested documents that were not included in the previously inserted document. The new JSON document defines a `platforms` key with an array of strings, with the platform names, in which the game can be executed.

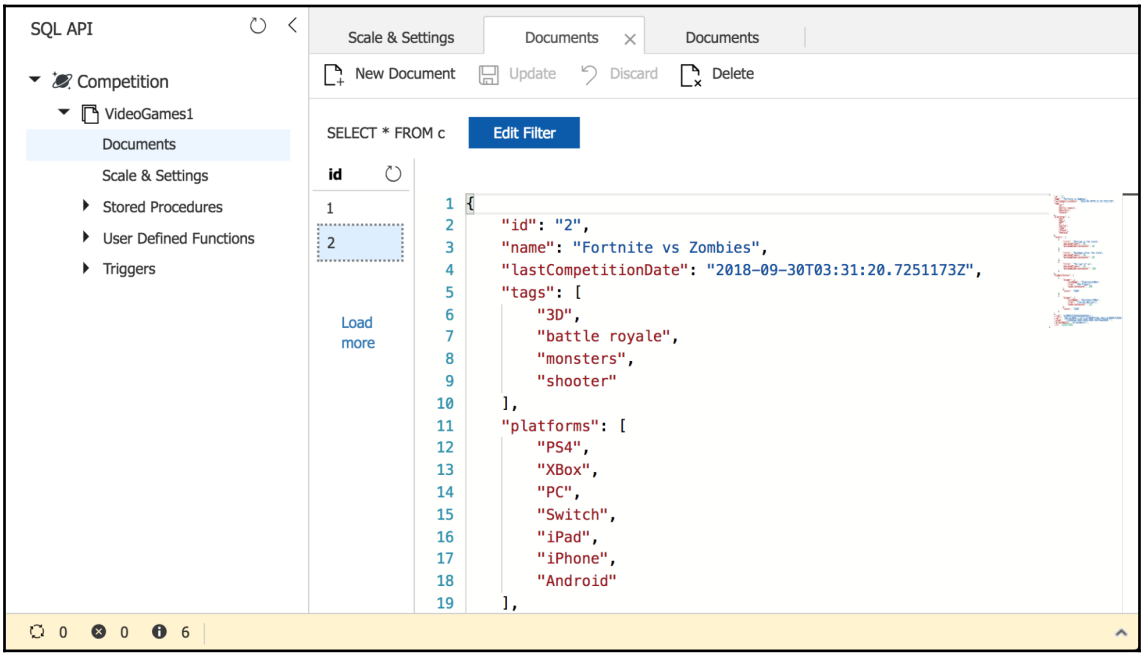
In addition, the new JSON document has different formats for the `levels` and `highestScores` arrays. In this case, each level has a title, a maximum number of players, and a minimum experience level. In this document, each high score adds an experience level that wasn't included in the previous document. We are going to add a new document that has a different schema to that of the previous document.

Copy the content of the new JSON document to the clipboard.

Click **New Document** on the toolbar below the **Documents** tab. The portal will display a JSON editor.

Select all the content in the editor and paste the content of the clipboard. This way, you will replace the previous lines with the new JSON document you created with your favorite text editor for JSON content.

Click **Save** on the toolbar below the **Documents** tab. The new document will be inserted into the `VideoGames1` collection and the portal will display the `id` for the new document in the list at the left-hand side and the inserted document in the editor at the right-hand side. The following screenshot shows the added document:



The document was successfully inserted, no matter the differences in the schema from the previous document that was added to the same collection. The document database with the SQL API is schema agnostic, and therefore we can have different schemas in each document we insert into the same collection without issues.

Scroll down to the latest lines of the document in the web-based editor and you will see the system-generated keys and their values, which we analyzed for the previously inserted document.

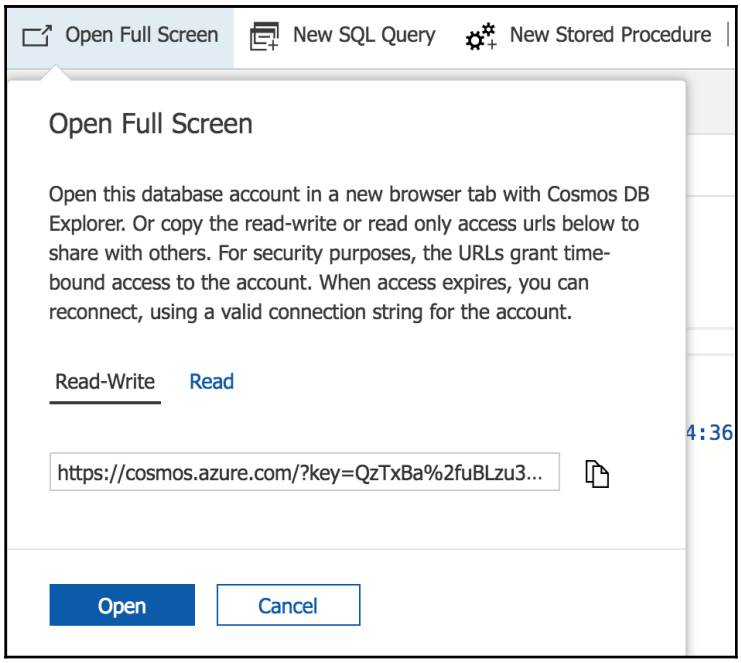
If you take a look at the value for the `_self` key for this document and compare it with the value of this key for the previous document, you will notice that they share the same prefix, because both documents belong to the same database and collection. The following are the two values I have after inserting the two documents. Note that the values will be different in your environment:

- Document with `id` equal to `1`:
  `"dbs/prUNAA==/colls/prUNAKtPjRo=/docs/prUNAKtPjRoFAAAAAAAAAA==/"`
- Document with `id` equal to `2`:
  `"dbs/prUNAA==/colls/prUNAKtPjRo=/docs/prUNAKtPjRoGAAAAAAAAAA==/"`

Working with the web-based Azure Cosmos DB Explorer

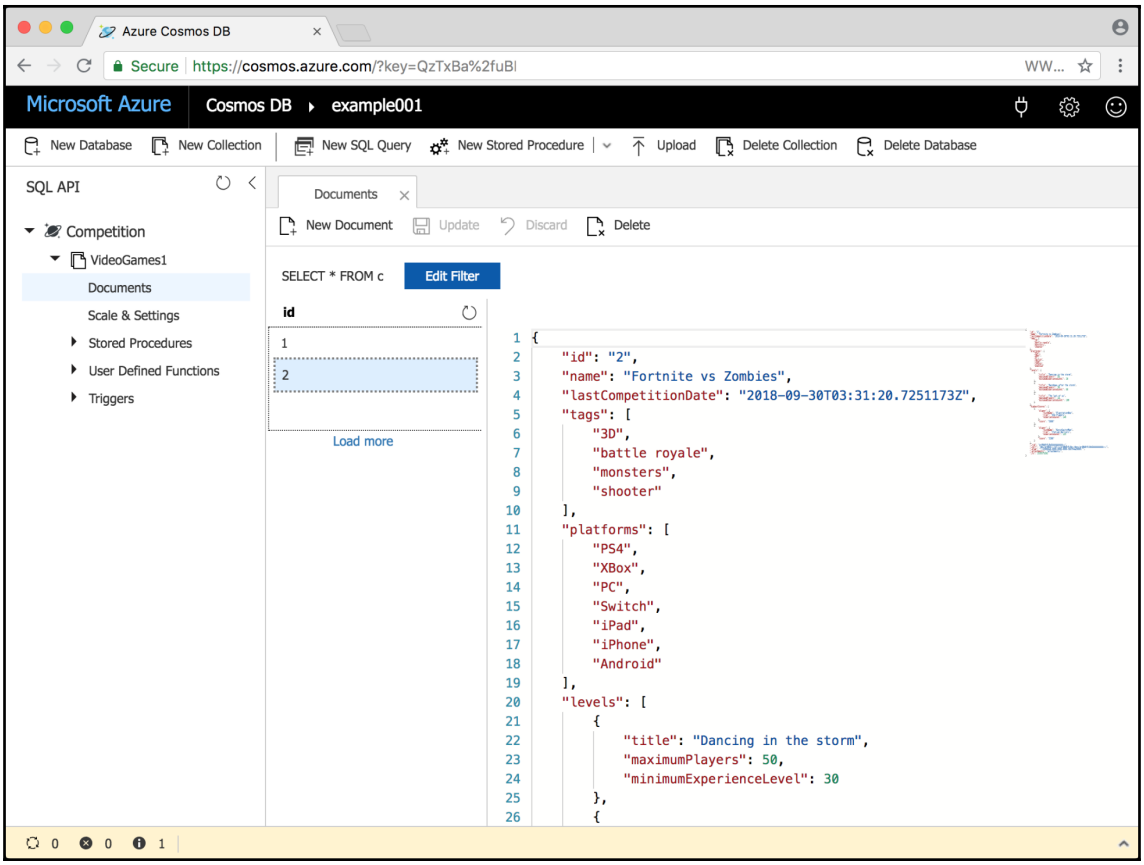
So far, we have been working with the portal and our screen real estate has been reduced as we selected new options that required additional panels. We can take advantage of Cosmos DB Explorer to open the Cosmos DB database account in a new browser tab and take advantage of a full-screen view.

Click **Open Full Screen** on the toolbar for the Cosmos DB database account. Cosmos DB will generate read-write and read-only access URLs that we can use to access the database and the collection in which we are working with Cosmos DB Explorer. The following screenshot shows a sample modal:



The URL will start with `https://cosmos.azure.com/?key=` followed by a key that will provide time-bound access to the Cosmos DB account with read-write or read-only access.

We just need to copy the URL and open it in a new browser tab. This way, we can work with Cosmos DB databases, collections, and documents with more screen real estate in any web browser. The following screenshot shows a web browser visualizing the previously created document, whose `id` is equal to `2`, with Cosmos DB Explorer:



We can take advantage of Cosmos DB Explorer to provide a temporary URL to other users who need to work with a Cosmos DB account, without having to create Azure users for them. In fact, these users don't need to have access to the Azure portal or a subscription. However, if you specify security policies for your collections and documents, these users won't be able to work with them.

Using Azure Storage Explorer to interact with Cosmos DB databases

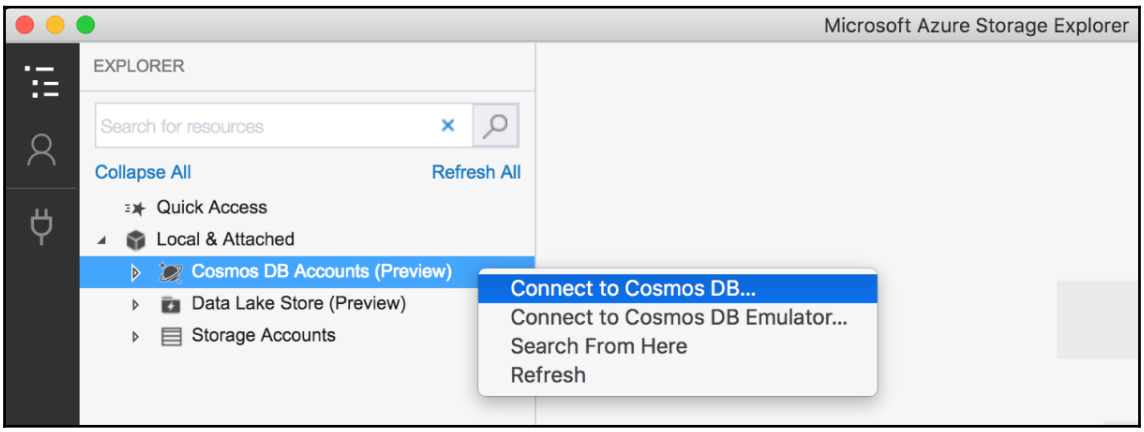
So far, we have been working with web-based interfaces to interact with a Cosmos DB Azure portal and Cosmos DB Explorer. The nice thing about web-based interfaces is that you can work with them in any compatible modern web browser on different devices. Now, we will explore a GUI tool that allows us to interact with Cosmos DB databases: Microsoft Azure Storage Explorer. Initially, this tool was intended to work with other Azure storage services. However, at the end of 2017, the tool added support for working with Cosmos DB databases.

This tool is available on Windows, macOS, and Linux. You can download and install Azure Storage Explorer here: `https://azure.microsoft.com/en-us/features/storage-explorer/`. Note that the instructions that I will provide were tested with version 1.4.1 and that you need to work with specific versions of the operating systems in order to be able to run the tool; this might vary in versions higher than 1.4.1.

At the time I was writing this book, this tool could only work with SQL API and Table API Cosmos DB accounts.

Once you have installed Azure Storage Explorer, launch the tool and you will see a panel titled **Explorer** located at the left-hand side with the **Cosmos DB Accounts** option within **Local & Attached**.

Right-click on **Cosmos DB Accounts** and the application will display a context menu, shown in the next screenshot:



Select **Connect to Cosmos DB...** and the application will open a dialog box to specify the necessary parameters for establishing a connection with a Cosmos DB account.

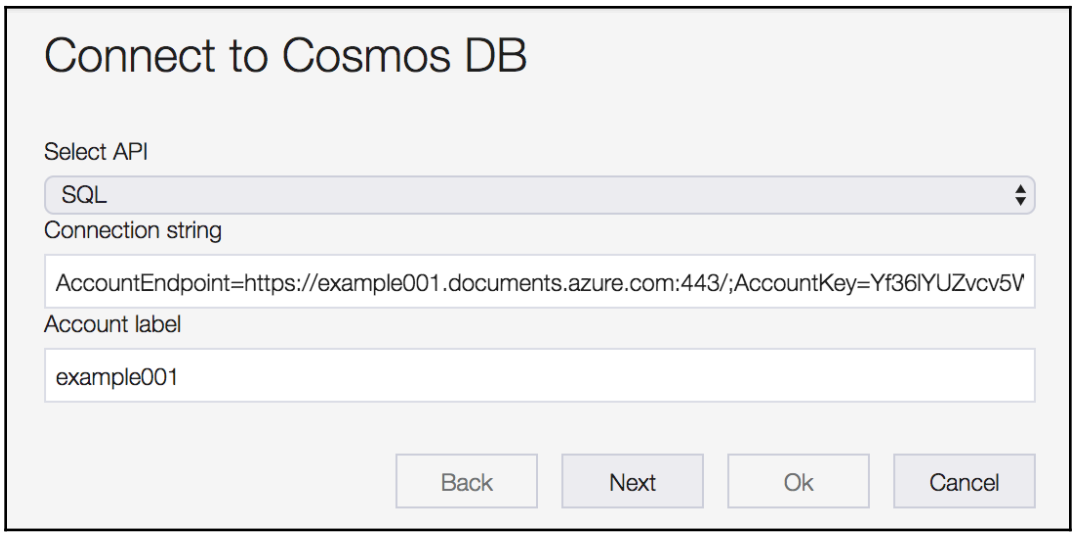
Select **SQL** in the **Select API** dropdown because we want to work with the Cosmos DB we created with the SQL API.

Copy the primary connection string we learned how to grab in the Azure portal in the *Understanding URIs, read-write and read-only keys, and connection strings* section of this chapter. Make sure you use the primary connection string, which enables read-write access to the account. The connection string starts with `AccountEndpoint=`.

Paste the primary connection string in the **Connection string** textbox.

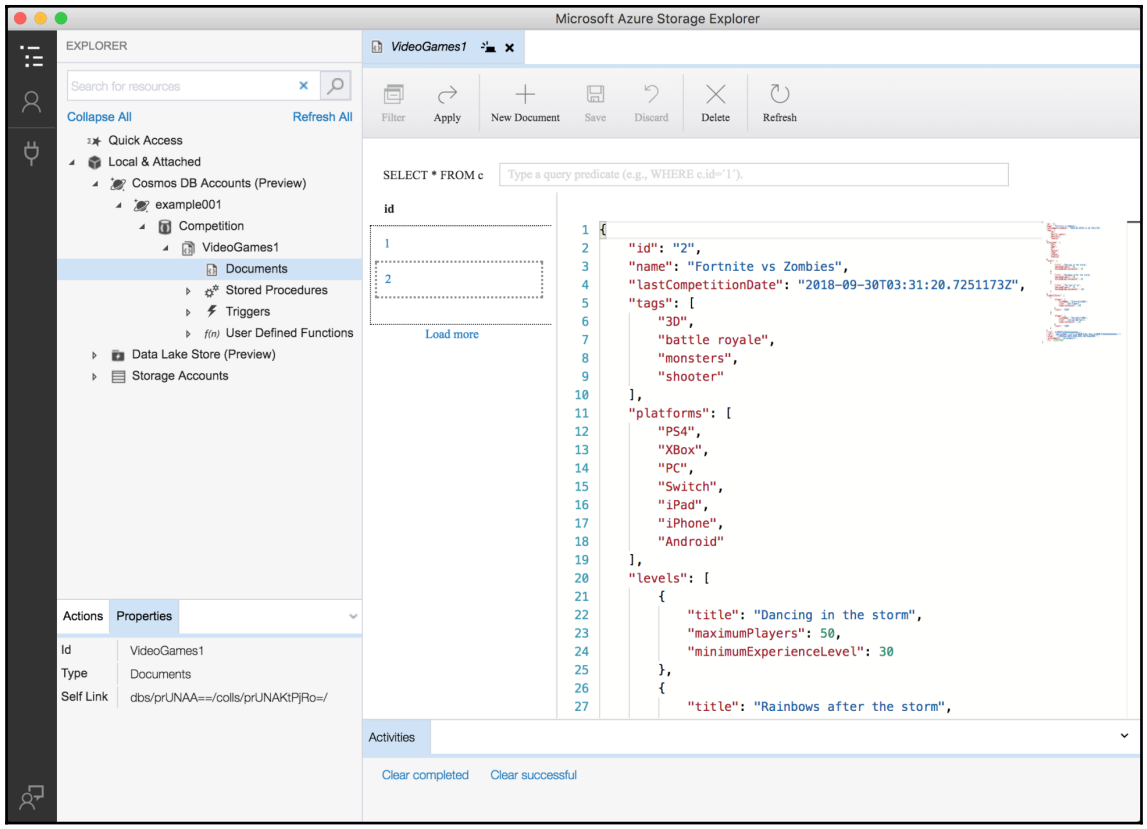
If you want to identify the Cosmos DB account with a different label than the proposed one, you can specify it in the **Account label** textbox.

The following screenshot shows an example of the configuration for the **Connect to Cosmos DB** dialog:



Click **Next** and the wizard will display the entered values so that you can confirm them. If they are correct, click **Connect**. Azure Storage Explorer will use the provided data to establish a connection with the Cosmos DB account and it will display the specified account label in the Cosmos DB Accounts list (`example001`) in the **Explorer** panel.

The application will allow you to navigate through the existing document database (`Competition`), its collection (`VideoGames1`), and its documents by expanding and clicking on the different elements. Once we reach the `Documents` element for the `VideoGames1` collection, we will see a panel that provides a toolbar and an interface that is very similar to the web-based interface we used in the Azure portal and in Cosmos DB Explorer. The following screenshot shows the application displaying the details of the document whose `id` is equal to `2`:



We can use Microsoft Azure Storage Explorer to perform the tasks we have been performing in the previous sections. The main difference is that we will see dialog boxes instead of the panels that we saw in the web-based interface.

Working with the Azure Cosmos DB Emulator

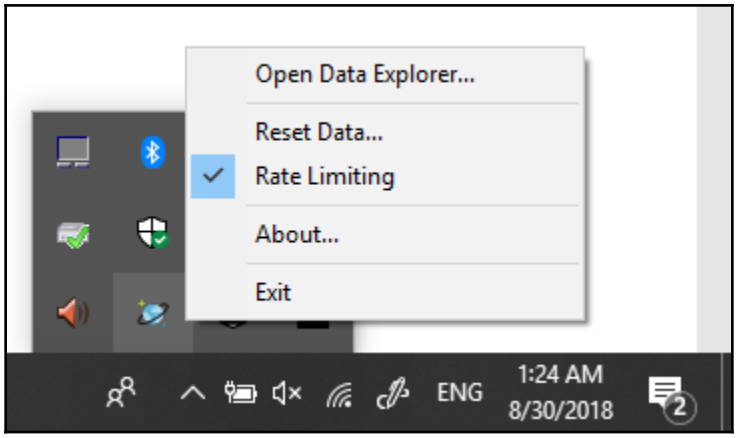
Sometimes, the Azure credit included in different subscriptions is enough for our development tasks. However, in other cases, we don't want to spend Azure credits or be billed for our development tasks that use Cosmos DB storage and services.

Microsoft provides the Azure Cosmos DB Emulator, which we can install on specific Windows versions, that emulates the Cosmos DB service without working with an Azure subscription or any other Azure service. The Azure Cosmos DB Emulator runs as an application that uses resources in the computer on which it is installed, and therefore makes it possible to perform many Cosmos DB operations and tests without incurring any Azure costs.

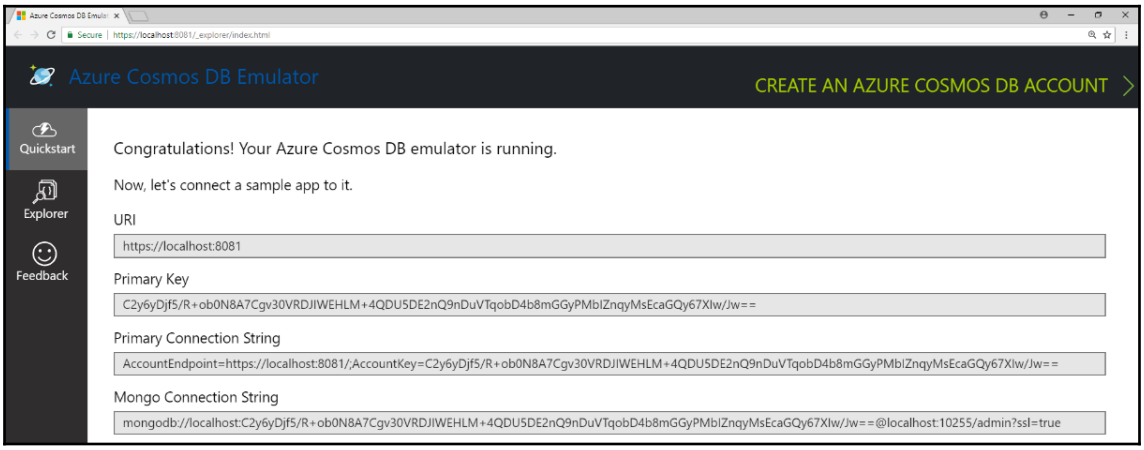
At the time I was writing this book, the emulator provided support for the SQL API and the MongoDB API. Other APIs were not available for use in the emulator data explorer. You can read more information about the emulator and download the latest binaries or Windows Docker images here: `https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator`. Unluckily, the emulator was only available for some specific Windows versions: Windows 10, Windows Server 2012 R2, Windows Server 2016, and Windows Server 2019.

After you install the emulator, you can work with its web-based data explorer or you can use the previously explained Microsoft Azure Storage Explorer to connect with it. You have to consider that even when the emulator is an active project and is continuously being improved, it has limitations on the number of collections and documents that you can create and doesn't have the same features that we have learned about so far for Cosmos DB. However, it will make it possible for us to run our sample applications. For example, you can perform all the tasks we have learned in the previous sections by working with the emulator.

After you execute the Azure Cosmos DB Emulator on Windows, right-click on its notification icon and select **Open Data Explorer...**, as shown in the next screenshot:



Your default web browser will go to `https://localhost:8081/_explorer/index.html` and you will see the **Quickstart** panel for the emulator, which will display the URI, primary key, and primary connection string for the SQL API. You can use these settings whenever you have to establish a connection to Cosmos DB and you will work with the emulator and your local resources instead of working with Azure Cosmos DB. We analyzed the meanings of these items in the *Understanding URIs, read-write and read-only keys, and connection strings* section of this chapter. In addition, the panel will display the connection string for the MongoDB API. The following screenshot shows the **Quickstart** panel:



You can access the data explorer for the emulator by clicking on **Explorer** in the left-hand panel. You will be able to create new databases, collections, and documents, as we did when working with the Azure portal.

If you work with the emulator, it is highly recommended to take advantage of the previously introduced Azure Storage Explorer. In order to connect with the emulator, launch the tool and you will see a panel titled **Explorer** located at the left-hand side with the **Cosmos DB Accounts** option in **Local & Attached**.

Select **Connect to Cosmos DB Emulator...** and the application will use the default parameters to establish a connection with the Azure Cosmos DB Emulator.

If you want to identify the Cosmos DB account with a different label than the proposed one, you can specify it in the **Account label** textbox. For example, you can specify `Emulator`.

Click **Next** and the wizard will display the entered values so that you can confirm them. If they are correct, click **Connect**. Azure Storage Explorer will use the provided data to establish a connection with the Azure Cosmos DB Emulator and it will display the specified account label in the Cosmos DB Accounts list (`Emulator`) in the **Explorer** panel.

The account that represents the emulator doesn't have a database yet, and therefore the first thing you will need to do is to create a database. Note that the account uses the default SQL API.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. If we want to store a JSON document in a Cosmos DB SQL API collection, which of the following keys must have a value to provide the required unique identifier:
    1. `id`
    2. `identifier`
    3. `uniqueId`
2. After a new document is added to a Cosmos DB SQL API collection, which of the following system-generated keys provides a unique addressable URI for the resource:
    1. `_selfLink`
    2. `_selfURI`
    3. `_self`
3. After a new document is added to a Cosmos DB SQL API collection, which of the following system-generated keys provides a timestamp with the last date and time at which the resource was updated:
    1. `_timeStamp`
    2. `_lastUpdateTS`
    3. `_ts`
4. We can establish a connection to a Cosmos DB account with:
    1. Only the Cosmos DB account URI or endpoint URL
    2. The Cosmos DB account URI or endpoint URL and the primary key as the authorization key
    3. The primary key and the secondary key concatenated to generate a single authorization key

5. Which of the following values specify that we want the value for the `name` key at the root level of a document to be unique per partition key:
    1. `+name`
    2. `name`
    3. `/name`

Summary

In this chapter, we provisioned a Cosmos DB account with the SQL API, and we created a document database and a collection. Then, we populated the collection with JSON documents and we understood the system-generated keys that Cosmos DB adds to a document resource. We ended up with a collection, containing documents with different structures, because we took advantage of the schema-agnostic feature of Cosmos DB.

We used the web-based Azure portal to perform the different tasks, and then we learned how to take advantage of screen real estate with Azure Cosmos DB Explorer. We worked with the Azure Storage Explorer GUI tool and we learned how to work with the Azure Cosmos DB Emulator to develop and test applications without being billed for the storage, request units, and bandwidth consumed.

Now that we have created our first Cosmos DB SQL API database, collection, and document, and we have explored the available tools, we will learn about building and running queries and taking advantage of indexing options, which are the topics we are going to discuss in the next chapter.

3
Writing and Running Queries on NoSQL Document Databases

In this chapter, we will write and run queries to retrieve data from documents in a collection. We will use the Cosmos DB dialect of SQL to work against a document database with the SQL API. We will understand the different ways of working with the documents, their sub-documents, and their arrays, and we will learn how queries consume resource units. We will do the following in this chapter:

- Run queries against a collection with different tools
- Understand query results in JSON arrays
- Check the request units spent by a query
- Work with schema-agnostic queries
- Use built-in array functions
- Work with joins
- Use array iteration
- Work with aggregate functions

Running queries against a collection with different tools

In `Chapter 2`, *Getting Started with Cosmos DB Development and NoSQL Document Databases*, we created an Azure Cosmos DB account with the SQL API, a document database, and a collection. Then, we inserted two JSON documents in the collection. Now, we will run queries against the collection with the two documents to learn the basics of the Cosmos DB dialect of SQL and how to run queries in the different web-based and GUI tools.

When Cosmos DB was launched, one of the most frustrating issues was that the only available tool to run queries was the web-based the Azure portal with its screen real estate problems. Luckily, Microsoft added the web-based Azure Cosmos DB Explorer and included support for Cosmos DB in the Azure Storage Explorer GUI tool. The three tools allow us to run queries against a collection by following very simple steps.

If you decide to work with the Azure Cosmos DB Emulator, you can use the web-based portal for the emulator or you can work with Azure Storage Explorer.

First, we will learn the necessary steps for running a simple query in each tool and to check the request units consumed by the query. Then, you can select the most appropriate tool for you to run the subsequent examples and then work with your databases.

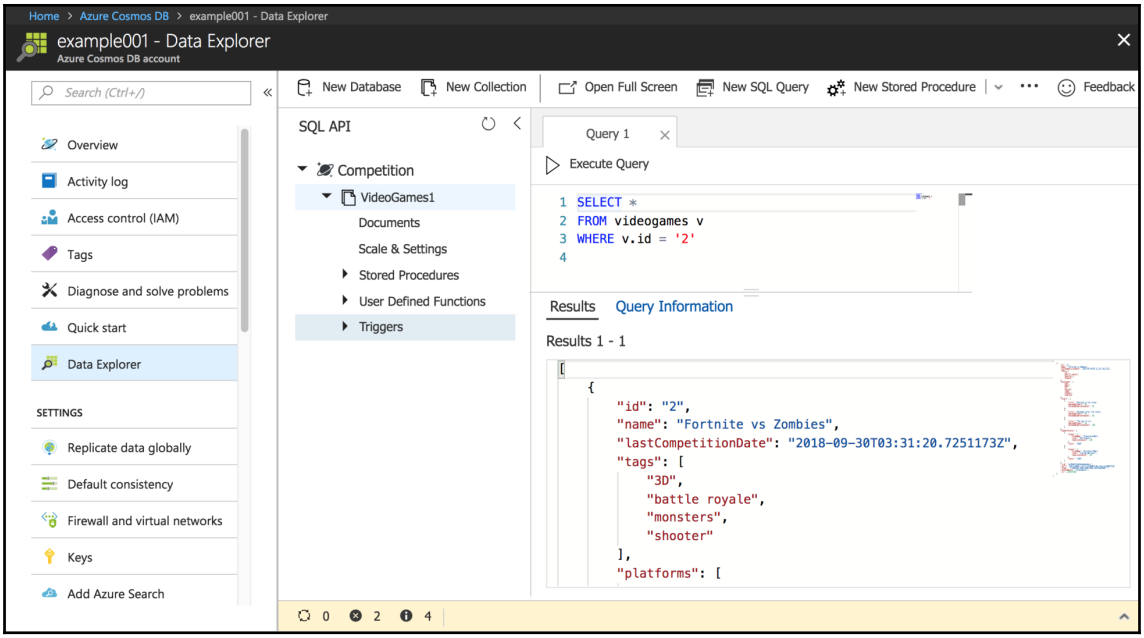
The following instructions will always avoid running the default filter, which retrieves documents for a collection whenever possible. This way, we avoid consuming unnecessary request units before running our desired query. However, some tools have a wrong design and don't allow us to write queries without retrieving the first documents as a first step. It is very important to understand that the filter feature included in the **Documents** view for a collection in the different tools doesn't provide the same result set that we will get from executing the same query. Thus, we won't apply filters by adding criteria in the filter panel found in the different tools. Instead, we will work with queries and their real results in order to be ready to work with the different SDKs.

Understanding query results in JSON arrays

The following lines show one of the simplest queries that we can run against the previously created `VideoGames1` collection and retrieves the document whose `id` is equal to `2`. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_01.sql` file:

```
SELECT *
FROM Videogames v
WHERE v.id = '2'
```

In the Azure portal, make sure you are in the page for the Cosmos DB account in the portal. Click on the **Data Explorer** option, click on the database name (`Competition`) to expand the collections for the database, and click on the collection name (`VideoGames1`). Click on **New SQL Query** in the toolbar located at the top of the panel and the portal will add a new query tab with a default query at the top: `SELECT * FROM c`. Select this text, paste the new text for the query, and click **Execute Query**. Cosmos DB will execute the query and the portal will display the JSON document with the results in the **Results** tab, as shown in the following screenshot:



Notice that the header for the results indicates that the **Results** panel is displaying the results from number 1 up to number 1 with the following label: **Results 1 - 1**. The following lines show the results of the query:

```
[
   {
        "id": "2",
        "name": "Fortnite vs Zombies",
        "lastCompetitionDate": "2018-09-30T03:31:20.7251173Z",
        "tags": [
            "3D",
            "battle royale",
            "monsters",
            "shooter"
        ],
        "platforms": [
            "PS4",
            "XBox",
            "PC",
            "Switch",
            "iPad",
            "iPhone",
            "Android"
        ],
        "levels": [
            {
                "title": "Dancing in the storm",
                "maximumPlayers": 50,
                "minimumExperienceLevel": 30
            },
            {
                "title": "Rainbows after the storm",
                "maximumPlayers": 30,
                "minimumExperienceLevel": 60
            },
            {
                "title": "The last of us",
                "maximumPlayers": 10,
                "minimumExperienceLevel": 100
            }
        ],
        "highestScores": [
            {
                "player": {
                    "nickName": "PlaystationBoy",
                    "clan": "USA Players",
                    "experienceLevel": 140
                },
```

```
                "score": "5600"
            },
            {
                "player": {
                    "nickName": "KevinSwitchMan",
                    "clan": "Italian Warriors",
                    "experienceLevel": 125
                },
                "score": "3300"
            }
        ],
        "_rid": "prUNAKtPjRoGAAAAAAAAAA==",
        "_self":
"dbs/prUNAA==/colls/prUNAKtPjRo=/docs/prUNAKtPjRoGAAAAAAAAAA==/",
        "_etag": "\"220066eb-0000-0000-0000-5b8704a20000\"",
        "_attachments": "attachments/",
        "_ts": 1535575202
    }
]
```

If we pay close attention to the previous JSON document returned as the result, we will notice that it is an array with the only JSON document that matches the query as its single element.

Whenever we execute a valid query in a collection with the SQL API, we will always receive a JSON array with one or more elements as a response. If our query just retrieves one document, we will receive this document as an element in an array, as in our current example. If we compare the results of our query with the results of inspecting the document that is stored in the database, we will notice the query results enclose the document in brackets (`[]`) to include the document as an element of an array. It is extremely important to understand the way the SQL API provides responses to queries.

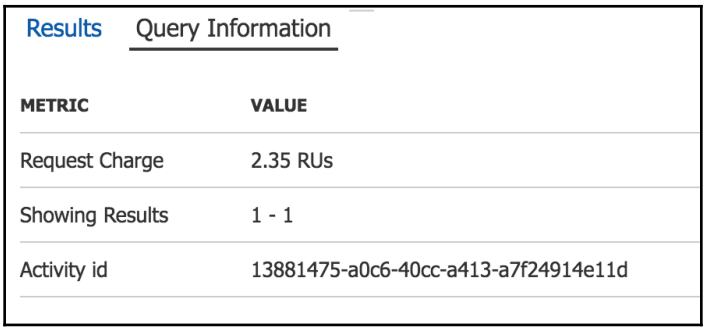
If you have experience with relational databases and their SQL dialects, you must consider that the following query isn't equivalent to the previously shown query and it has an invalid syntax for the Cosmos DB SQL dialect:

```
SELECT v.*
FROM Videogames v
WHERE v.id = '2'
```

Hence, whenever you want to retrieve all the contents, just use `SELECT *` without the alias followed by a dot (`.`) as a prefix.

Checking the request units spent by a query

Every query we execute consumes request units. We can easily check the request charges for a query by clicking on the **Query Information** tab. The following screenshot shows the information provided by this tab for the previously executed query:



The value for the **Request Charge** metric specifies the request units that we were charged by Cosmos DB for the executed query. In this case, the query spent 2.35 request units from the request units we are provisioned for the `VideoGames1` collection. Remember that we configured the settings for this collection to provide a throughput of 1,000 request units per second. Hence, after we execute this query, we will have 1,000 - 2.35 = 997.65 request units remaining after they are reset to 1,000 in the next second. In an application, we would be able to run this query 425 times in one second with the provisioned 1,000 request units per second. We would have to wait until the next second to run this query again in order to have 1,000 request units available for one second.

In this case, we are inspecting the query information in the Azure portal. It is very important to know that whenever we execute queries through the different Cosmos DB SDKs, we also receive this information as part of the results, and therefore, we are able to check the request units spent by a query. In fact, the SDKs provide a huge amount of additional data related to the query execution. Unluckily, at the time I was writing the book, the Azure portal didn't offer more information about the query execution.

The Cosmos DB account is using the default consistency level for database accounts: session consistency. Notice that the selected default consistency level has an impact on the request units charged to a query.

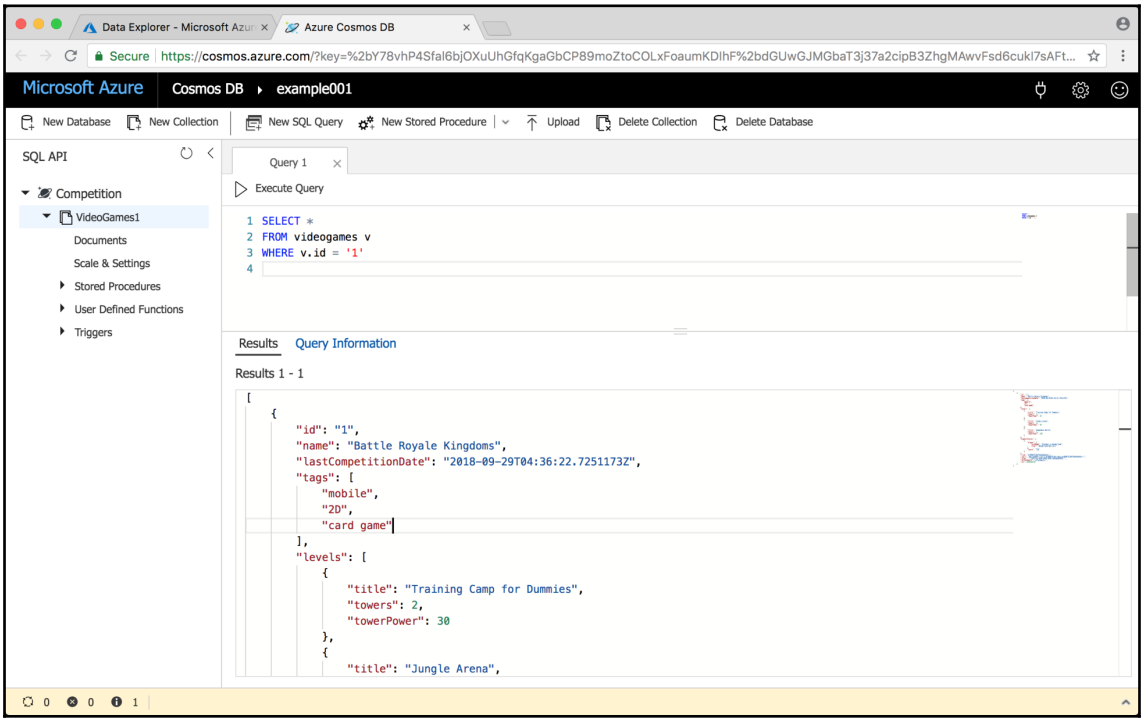
Now, let's make a simple change to the previous query to retrieve a different document. Specifically, we will indicate we want to retrieve the document whose `id` is equal to `1`. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_02.sql` file:

```
SELECT *
FROM Videogames v
WHERE v.id = '1'
```

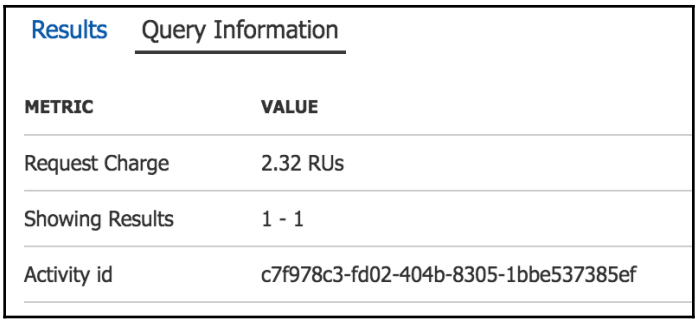
This time, we will take advantage of the Cosmos DB Explorer to run the query and benefit from a full-screen view:

1. Click **Open Full Screen** on the toolbar for the Cosmos DB database account and then click **Open** to open the read-write URL in a new tab. The left-hand panel will list all the connections for the `Competition` database.
2. Click on the collection name (`VideoGames1`). Click on **New SQL Query** in the toolbar located at the top of the panel and the portal will add a new query tab with a default query at the top: `SELECT * FROM c`.

3. Select this text, paste the new text for the query, and click **Execute Query**. Cosmos DB will execute the query and the portal will display the JSON document with the results in the **Results** tab, as shown in the next screenshot. Notice the additional screen real estate compared with the usage of the Azure portal for the same task:



4. Click on the **Query Information** tab to check the request charges for this query. The following screenshot shows the information provided by this tab for the recently executed query:



The new query has a request charge of 2.32 request units, which is lower than the 2.35 request units spent by the previous query. The only difference between the two queries is the `id` value. The new query consumes 0.03 fewer request units because the retrieved document that is returned as the single element of the result array is smaller than the document whose `id` is equal to `2`. The document retrieved by the new query has 1,180 characters and the document retrieved by the first query has 1,731 characters.

The amount of data retrieved by a query has an impact on the request units charged to it by Cosmos DB.

Working with schema-agnostic queries

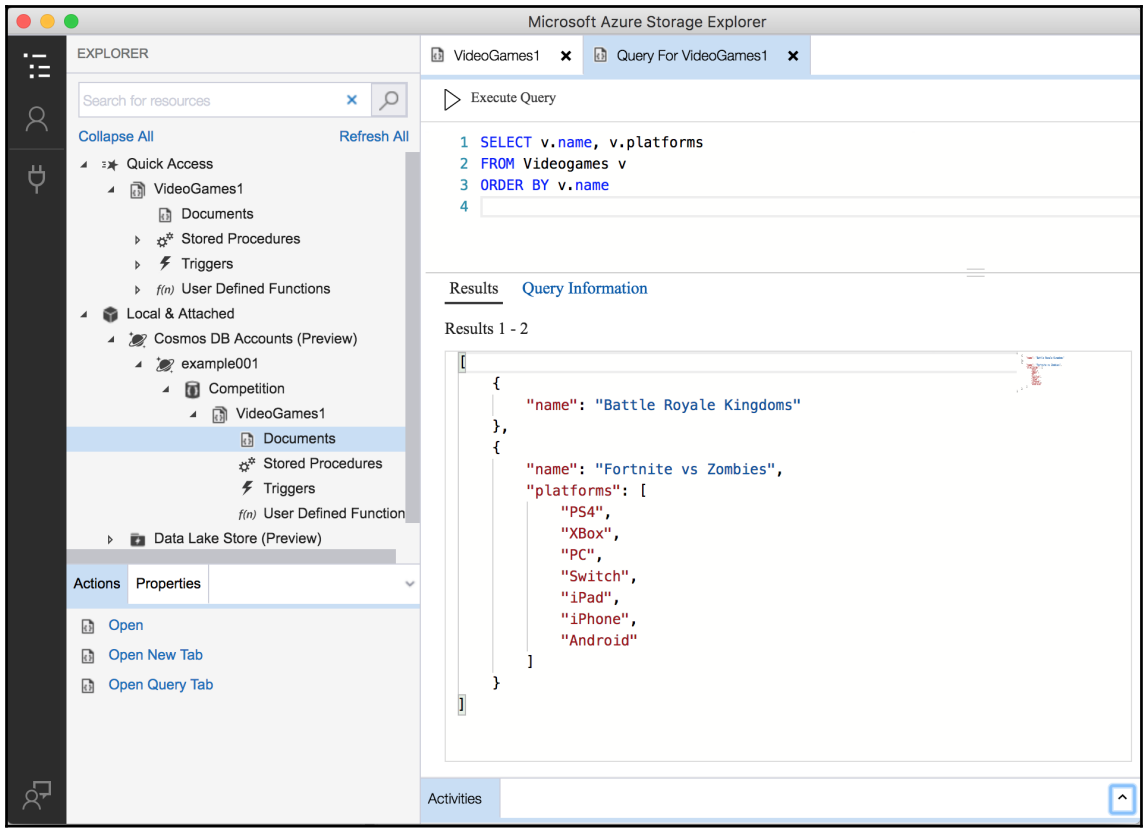
The `VideoGames1` collection has two documents. The document with `id` equal to `2` has a `platforms` key whose value is an array of string with the platforms in which the video game can be executed. The document with `id` equal to `1` doesn't include the `platforms` key.

Now we will write a query that will indicate the properties we want to retrieve from all the documents in the collection. Specifically, we will specify we want to retrieve the `name` and `platforms` properties. In addition, we will request the results to be sorted by name in ascending order. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_03.sql` file:

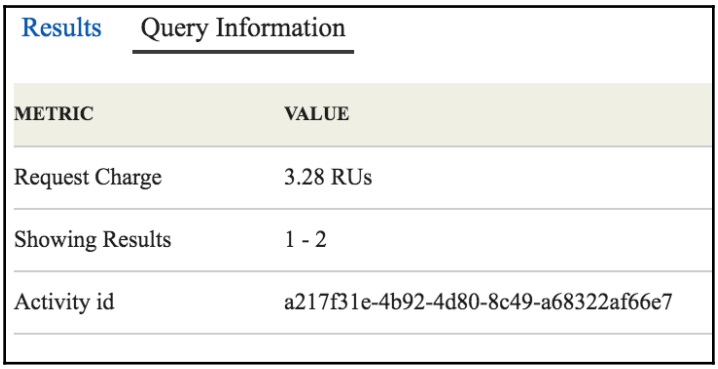
```
SELECT v.name,
    v.platforms
FROM Videogames v
ORDER BY v.name
```

You can follow the steps to execute the preceding query:

1. Go to Azure Storage Explorer and navigate through the existing document database (`Competition`), its collection (`VideoGames1`), and its `Documents` element by expanding and clicking on the different elements.
2. Right-click on the `Documents` element for the `VideoGames1` collection and select the **Open query** tab in the context menu. The application will open a new query tab with a default query at the top: `SELECT * FROM c`.
3. Select this text, paste the new text for the query, and click **Execute Query**. Cosmos DB will execute the query and the application will display the JSON document with the results in the **Results** tab, as shown in the following screenshot:



4. As happened with the other tools, we can click on the **Query Information** tab to check the request charges for this query. The following screenshot shows the information provided by this tab for the recently executed query:


Results Query Information

METRIC	VALUE
Request Charge	3.28 RUs
Showing Results	1 - 2
Activity id	a217f31e-4b92-4d80-8c49-a68322af66e7

The result for this query is an array with two elements that have different keys or properties. The first element only provides the value for the `name` key because the document whose `id` is equal to `1` didn't have a key named `platforms` at the root level. The second element provides the value for the `name` and `platform` keys. The query takes advantage of the schema-agnostic features in Cosmos DB and the document database, and therefore, we can request properties that aren't present in all of the documents and the query provides the results without issues.

It is very important to understand that the element related to the document that doesn't have a requested property won't include the key in the result.

This query required the retrieval of two documents, the extraction of the required properties from them, and the sorting of the results by the `name` property in ascending order. In this case, the request charge for the query is `3.28` request units.

We have learned the necessary steps to execute queries on the different available tools. Now we will focus on running useful queries and analyzing their results. You just need to follow the previously explained steps in your favorite tool to check the results.

Cosmos DB supports many built-in functions, including type-checking functions. For example, the following query retrieves the `id` of the documents that define the `platforms` key by taking advantage of the `IS_DEFINED` built-in type-checking function. This function returns a Boolean indicating whether the property received as an argument has been assigned a value. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_04.sql` file:

```
SELECT v.id
FROM Videogames v
WHERE IS_DEFINED(v.platforms)
```

The following lines show the results of the query. Only the video game whose `id` is equal to `"2"` defined the `platforms` property:

```
[
    {
        "id": "2"
    }
]
```

The two documents in the `VideoGames1` collection have the `levels` key with an array of JSON documents that defines the different levels that the game has. In the two video games, the properties that define a level have different properties and only the `title` property is included in the levels for both video games. The video game whose `id` is equal to `1` defines a level with the following properties: `title`, `towers`, and `towerPower`. The video game whose `id` is equal to `2` defines a level with the following properties: `title`, `maximumPlayers`, and `minimumExperienceLevel`.

The following query will build a new document for the first level of each video game document that has a `levels` property defined at the root level and its value is an array. The query takes advantage of the `IS_ARRAY` built-in type-checking function. This function returns a Boolean indicating whether the property received as an argument has been assigned a value. In addition, the query uses the `AS` keyword to generate a property based on the expression specified at the left-hand side. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_05.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels[0] AS firstLevel
FROM videogames v
WHERE IS_ARRAY(v.levels)
```

The `v.id` and `v.name` properties are mapped to the `videoGameId` and `videoGameName` properties. The first level in the `levels` array (`v.levels[0]`) is mapped to the `firstLevel` property.

We can access the elements of an array with the desired element enclosed within brackets (`[]`) after the property name, with a syntax that is very common in many programming languages. Notice that the arrays use `0` for the index origin, and therefore, the first element in an array is addressed with `[0]`.

The following lines show the results of the query. Notice that the `firstLevel` property has an object with different properties in each case:

```
[
    {
        "videoGameId": "1",
        "videoGameName": "Battle Royale Kingdoms",
        "firstLevel": {
            "title": "Training Camp for Dummies",
            "towers": 2,
            "towerPower": 30
        }
    },
    {
        "videoGameId": "2",
        "videoGameName": "Fortnite vs Zombies",
        "firstLevel": {
            "title": "Dancing in the storm",
            "maximumPlayers": 50,
            "minimumExperienceLevel": 30
        }
    }
]
```

In some cases, we will want to limit the number of values returned in the query. The `TOP` operator followed by a number will do this job. The following lines show a new version of the previous query that will sort the results by the video game's name and will return an array with only the first document of all the video games that match the specified criteria. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_06.sql` file:

```
SELECT TOP 1 v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels[0] AS firstLevel
FROM videogames v
WHERE IS_ARRAY(v.levels)
ORDER BY v.name
```

The following lines show the results of the query:

```
[
    {
        "videoGameId": "1",
        "videoGameName": "Battle Royale Kingdoms",
        "firstLevel": {
            "title": "Training Camp for Dummies",
            "towers": 2,
            "towerPower": 30
        }
    }
]
```

Using built-in array functions

The previous query made sure that it only processes each `videogame` document that has a `level` property defined at the root level and its value is an array. However, it is possible to have a new `videogame` document that declares an empty array (`[]`) as the value for the `level` property because the video game doesn't have defined levels yet.

The next query is a new version of the previous query that takes advantage of the `ARRAY_LENGTH` built-in function. This function returns the number of elements of the array expression received as an argument. The query makes sure that the `level` property is an array and that it contains at least one element. The results for the query will be the same that were shown for its previous version. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_07.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels[0] AS firstLevel
FROM Videogames v
WHERE IS_ARRAY(v.levels)
AND ARRAY_LENGTH(v.levels) >= 1
```

The following query will build a new document for each video game that has at least three elements in the `levels` array at the root level. The query takes advantage of the `ARRAY_SLICE` built-in function. This function returns the part of the array expression received as an argument specified by the starting element and the number of elements. The query uses this function in combination with the `AS` keyword to generate the `selectedLevels` property with an array composed of the second and third elements of the `levels` array. Notice that the parameters for `ARRAY_SLICE` after `v.levels` are `1` and `2`, which means we want the part of the array starting with the second element (the first element would be `0`) and we must extract `2` elements. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_08.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    ARRAY_SLICE(v.levels, 1, 2) AS selectedLevels
FROM Videogames v
WHERE IS_ARRAY(v.levels)
AND (ARRAY_LENGTH(v.levels) >= 3)
```

The following lines show the results of the query. Notice that the `selectedLevels` property has an array with objects with different properties in each case:

```
[
    {
        "videoGameId": "1",
        "videoGameName": "Battle Royale Kingdoms",
        "selectedLevels": [
            {
                "title": "Jungle Arena",
                "towers": 2,
                "towerPower": 40
```

```
            },
            {
                "title": "Legendary World",
                "towers": 5,
                "towerPower": 100
            }
        ]
    },
    {
        "videoGameId": "2",
        "videoGameName": "Fortnite vs Zombies",
        "selectedLevels": [
            {
                "title": "Rainbows after the storm",
                "maximumPlayers": 30,
                "minimumExperienceLevel": 60
            },
            {
                "title": "The last of us",
                "maximumPlayers": 10,
                "minimumExperienceLevel": 100
            }
        ]
    }
]
```

The two documents in the `VideoGames1` collection have the `tags` key with an array of strings with tags related to the video game. The following query will build a new document for each videogame that includes `monsters` as one of the string values of the `tags` key. The query takes advantage of the `ARRAY_CONTAINS` built-in function. This function returns a Boolean indicating whether the array received as an argument contains the specified value. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_09.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.tags AS videoGameTags
FROM Videogames v
WHERE ARRAY_CONTAINS(v.tags, "monsters")
```

The following lines show the results of the query. Only one document includes `"monsters"` as one of the string values of the `tags` key:

```
[
    {
        "videoGameId": "2",
        "videoGameName": "Fortnite vs Zombies",
```

```
            "videoGameTags": [
                "3D",
                "battle royale",
                "monsters",
                "shooter"
            ]
        }
    ]
```

Now we want to retrieve some properties for each video game that includes a specific level definition in the `levels` array. The following query will build a new document for each video game that includes a level definition with specific `title`, `maximumPlayers`, and `minimumExperienceLevel` values. The query uses the previously introduced `ARRAY_CONTAINS` built-in function. This time, the query calls this function with an object as the second argument. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_10.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels AS videoGameLevels
FROM Videogames v
WHERE ARRAY_CONTAINS(v.levels,
    {
        "title": "Rainbows after the storm",
        "maximumPlayers": 30,
        "minimumExperienceLevel": 60
    })
```

The following lines show the results of the query. Only one document includes the requested level definition as one of the objects of the `levels` key:

```
[
    {
        "videoGameId": "2",
        "videoGameName": "Fortnite vs Zombies",
        "videoGameLevels": [
            {
                "title": "Dancing in the storm",
                "maximumPlayers": 50,
                "minimumExperienceLevel": 30
            },
            {
                "title": "Rainbows after the storm",
                "maximumPlayers": 30,
                "minimumExperienceLevel": 60
            },
            {
```

```
                "title": "The last of us",
                "maximumPlayers": 10,
                "minimumExperienceLevel": 100
            }
        ]
    }
]
```

Now we will make some changes to the previous query and we will specify a level definition with specific `title` and `towers` values. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_11.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels AS videoGameLevels
FROM Videogames v
WHERE ARRAY_CONTAINS(v.levels,
    {
        "title": "Jungle Arena",
        "towers": 2
    })
```

The following line shows the results of the query, an empty array:

```
[]
```

One of the documents includes the following level definition as one of the objects of the `levels` key. However, the query result is an empty array because we are using the default option for the Boolean third argument of the `ARRAY_CONTAINS` built-in function, which specifies whether we want a partial match. The default value for this argument is `false` and it indicates that we want all the elements to match, and therefore, only an object that provides exactly the same key-value pairs that are specified will evaluate to `true`:

```
{
    "title": "Jungle Arena",
    "towers": 2,
    "towerPower": 40
}
```

The following is a new version of the query that specifies `true` as the third argument for the `ARRAY_CONTAINS` built-in function and indicates that we are specifying a partial fragment that we want to match. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_12.sql` file:

```
SELECT v.id AS videoGameId,
    v.name AS videoGameName,
    v.levels AS videoGameLevels
FROM Videogames v
WHERE ARRAY_CONTAINS(v.levels,
    {
        "title": "Jungle Arena",
        "towers": 2
    }, true)
```

The following lines show the results of the query. Only one document includes a partial match for the requested fragment of the level definition as one of the objects of the `levels` key:

```
[
    {
        "videoGameId": "1",
        "videoGameName": "Battle Royale Kingdoms",
        "videoGameLevels": [
            {
                "title": "Training Camp for Dummies",
                "towers": 2,
                "towerPower": 30
            },
            {
                "title": "Jungle Arena",
                "towers": 2,
                "towerPower": 40
            },
            {
                "title": "Legendary World",
                "towers": 5,
                "towerPower": 100
            }
        ]
    }
]
```

Working with joins

The two documents in the `VideoGames1` collection have the `highestScores` key with an array of JSON documents that provide a `player` object, also known as sub-document when we work with Cosmos DB. The array of JSON documents also provide a `score` value. In the two video games, the player sub-documents have different properties and only the `nickName` and `clan` properties are included in the `player` sub-document for the high score definitions of both video games. The video game whose `id` is equal to `"1"` defines a player with the following properties: `nickName` and `clan`. The video game whose `id` is equal to `2` defines a player with the following properties: `nickName`, `clan`, and `experienceLevel`.

Now, we want to retrieve the video game name, the player's nickname, and the high score achieved by the player on the game. The following table summarizes the properties we want to retrieve and the alias that we will assign to each of them:

Property	**Alias**
`name`	`videoGameName`
`highestScore` → `player` → `nickName`	`playerNickName`
`highestScore` → `score`	`highScore`

The following query will build a cross product of the video game documents and their high scores sub-documents by using a **self-join**. The query will take advantage of the `JOIN` clause, which forms tuples from the full cross product of specified sets. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_13.sql` file:

```
SELECT v.name AS videoGameName,
    h.player.nickName AS playerNickName,
    h.score AS highScore
FROM Videogames v
JOIN h IN v.highestScores
```

The query produces the full product of the video game documents (the root documents) with the `highestScores` sub-document for each game and assigns the `h` alias to each `highScore` sub-document. The `JOIN h IN v.highestScores` clause generates an iterator that expands each child element, `h`, in the `v.highestScores` array and applies a cross product with the root of the document whose alias is `v` with each flattened child element, `h`. The query projects the desired properties from the cross product.

The following lines show a pseudo-code that rewrites the previous query with imperative code:

```
resultArray = [];
foreach (var v in Videogames)
{
    foreach (var h in v.highestScores)
    {
        resultTuple = new Tuple(
            videoGameName: v.name,
            highScore: h.player.nickName,
            highScore : h.score);
        resultArray.Add(resultTuple);
    }
}
return resultArray;
```

The following lines show the results of the query. There is one document per high score registered. The first document has one high score and the second document has two high scores, and therefore, the results array has three documents (2 + 1 = 3):

```
[
    {
        "videoGameName": "Battle Royale Kingdoms",
        "playerNickName": "Brandon in Wonderland",
        "highScore": "750"
    },
    {
        "videoGameName": "Fortnite vs Zombies",
        "playerNickName": "PlaystationBoy",
        "highScore": "5600"
    },
    {
        "videoGameName": "Fortnite vs Zombies",
        "playerNickName": "KevinSwitchMan",
        "highScore": "3300"
    }
]
```

It is possible to chain multiple joins. However, it is very important to understand that we are asking Cosmos DB to produce full cross products. We must be extremely careful with self-joins.

Using array iteration

Now we will build a simple query to retrieve all the high scores defined for each video game flattened into a single array. The following query will take advantage of the `IN` keyword, which makes it possible to iterate over properties that are of the array type. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_14.sql` file:

```
SELECT *
FROM h IN Videogames.highestScores
```

The following lines show a pseudo-code that rewrites the previous query with imperative code:

```
resultArray = [];
foreach (var v in Videogames)
{
    foreach (var h in v.highestScores)
    {
        resultArray.Add(h);
    }
}
return resultArray;
```

The following lines show the results of the query. Notice that each highest score document is an element of the generated array:

```
[
    {
        "player": {
            "nickName": "Brandon in Wonderland",
            "clan": "Wonderland Warriors"
        },
        "score": "750"
    },
    {
        "player": {
            "nickName": "PlaystationBoy",
            "clan": "USA Players",
            "experienceLevel": 140
        },
        "score": "5600"
    },
    {
        "player": {
            "nickName": "KevinSwitchMan",
            "clan": "Italian Warriors",
```

```
            "experienceLevel": 125
        },
        "score": "3300"
    }
]
```

Now, we will add a filter to retrieve all the high scores defined for each video game flattened into a single array that have a player object with an experience level higher than `120`. The next query combines the use of the `IN` keyword with a filter on the flattened single array it generates. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_15.sql` file:

```
SELECT *
FROM h IN Videogames.highestScores
WHERE h.player.experienceLevel > 120
```

The following lines show a pseudo-code that rewrites the previous query with imperative code:

```
resultArray = [];
foreach (var v in Videogames)
{
    foreach (var h in v.highestScores)
    {
        if (h.player.experienceLevel > 120)
        {
            resultArray.Add(h);
        }
    }
}
return resultArray;
```

The following lines show the results of the query. Notice that each highest score document whose `player` has an `experienceLevel` value higher than `120` is an element of the generated array:

```
[
    {
        "player": {
            "nickName": "PlaystationBoy",
            "clan": "USA Players",
            "experienceLevel": 140
        },
        "score": "5600"
    },
    {
        "player": {
```

```
            "nickName": "KevinSwitchMan",
            "clan": "Italian Warriors",
            "experienceLevel": 125
        },
        "score": "3300"
    }
]
```

It is extremely important to understand the difference between the use of the `IN` operator in a query and a similar query that produces different results because it doesn't flatten the highest scores in a single array. Thus, we will analyze the next queries in detail.

The following query retrieves all the `highestScores` arrays for each video game and adds each of them to the results array. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_16.sql` file:

```
SELECT *
FROM Videogames.highestScores h
```

The following lines show a pseudo-code that rewrites the previous query with imperative code:

```
resultArray = [];
foreach (var v in Videogames)
{
    resultArray.Add(v.highestScores);
}
return resultArray;
```

The following lines show the results of the query. Notice that each `highestScore` array is an element of the generated array. When we used the `IN` keyword, each highest score document was an element of the generated array. Of course, in this case, there is a big difference with the previous query, in that we don't use a `WHERE` clause and the results are different:

```
[
    [
        {
            "player": {
                "nickName": "Brandon in Wonderland",
                "clan": "Wonderland Warriors"
            },
            "score": "750"
        }
    ],
    [
        {
```

```
            "player": {
                "nickName": "PlaystationBoy",
                "clan": "USA Players",
                "experienceLevel": 140
            },
            "score": "5600"
        },
        {
            "player": {
                "nickName": "KevinSwitchMan",
                "clan": "Italian Warriors",
                "experienceLevel": 125
            },
            "score": "3300"
        }
    ]
]
```

Working with aggregate functions

Cosmos DB SQL provides support for aggregations in the `SELECT` clause. For example, the following query uses the `SUM` aggregate function to sum all the values in the expression and calculate the total number of levels in the filtered games. The query uses the `ARRAY_LENGTH` built-in function to calculate the length of the levels array for each game and use it as an argument for the `SUM` aggregate function. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_17.sql` file:

```
SELECT SUM(ARRAY_LENGTH(v.levels))
FROM Videogames v
```

The following lines show the results of the query. Notice that the element of the array includes a key named `$1`:

```
[
    {
        "$1": 6
    }
]
```

Whenever we use an expression in the `SELECT` clause that is not a property name and we don't specify the desired alias name, Cosmos DB generates a key that starts with the `$` prefix and continues with a number that starts in `1`. Hence, if we have three expressions that aren't property names and don't include their desired aliases, Cosmos DB will use `$1`, `$2` and `$3` for the properties in the output results.

If we just want to generate the value without a key in the result, we can use the `VALUE` keyword. The following query uses this keyword. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_18.sql` file:

```
SELECT VALUE SUM(ARRAY_LENGTH(v.levels))
FROM Videogames v
```

The following lines show the results of the query. Notice that the element of the array doesn't include the key:

```
[
    6
]
```

It is also possible to achieve the same goal by using the `COUNT` aggregate function combined with the `IN` keyword. The following query uses the `COUNT` aggregate function to count the number of items in the expression and calculate the total number of levels in the iterated levels for all the games. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_19.sql` file:

```
SELECT COUNT(l) AS totalNumberOfLevels
FROM l IN Videogames.levels
```

The following lines show the results of the query. Notice that, in this case, the query specified the desired alias:

```
[
    {
        "totalNumberOfLevels": 6
    }
]
```

Now we want to calculate the average tower power for the levels defined in the video games. The `towerPower` property is not defined for all the levels and it is only available for the levels of the game whose `id` is equal to `1`. Whenever we use the `AVG` aggregate function to calculate an average for an expression, only the documents that have the property will be part of the average calculation. Hence, the levels that don't have the `towerPower` property won't generate an impact on the average. The following query uses the `AVG` aggregate function combined with the `IN` keyword to iterate all the levels of the games that have the `towerPower` property and compute its average value. The code file for the sample is included in the `learning_cosmos_db_03_01` folder in the `sql_queries/videogame_1_20.sql` file:

```
SELECT AVG(l.towerPower) AS towerPowerAverage
FROM l IN Videogames.levels
```

The following lines show the results of the query. Notice that, in this case, the query specified the desired alias:

```
[
    {
        "towerPowerAverage": 56.666666666666664
    }
]
```

In order to make things simpler, we have been always running queries that used the default indexing and worked with a single partition. In the next chapter, we will analyze indexing strategies for our application and we will work with multiple partitions.

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. Which of the following queries is valid in the Cosmos DB SQL dialect?
    1. `SELECT g.* FROM Games g WHERE g.id == '5'`
    2. `SELECT g.* FROM Games g WHERE g.id = '5'`
    3. `SELECT * FROM Games g WHERE g.id = '5'`

2. The `IS_DEFINED` built-in type-checking function does what exactly?
    1. Returns a Boolean indicating whether the property received as an argument has been assigned a value
    2. Returns the number of times a property has been assigned a value in the array expression received as an argument
    3. Returns a Boolean indicating whether the property received as an argument has been defined as required for the collection that contains the document
3. Which of the following queries retrieves all the levels flattened into a single array?
    1. `SELECT * FROM l IN Games.levels`
    2. `SELECT FLAT * FROM Games.levels`
    3. `SELECT * FROM Games.levels[0]`
4. Which of the following queries returns an array with a single value without a key?
    1. `SELECT TOP 1 SUM(ARRAY_LENGTH(g.highestScores)) FROM Games g`
    2. `SELECT FLAT SUM(ARRAY_LENGTH(g.highestScores)) FROM Games g`
    3. `SELECT VALUE SUM(ARRAY_LENGTH(g.highestScores)) FROM Games g`
5. Which of the following queries returns an array with a key-value pair?
    1. `SELECT TOP 1 VALUE SUM(ARRAY_LENGTH(g.highestScores)) FROM Games g`
    2. `SELECT SUM(ARRAY_LENGTH(g.highestScores)) AS totalHighestScores FROM Games g`
    3. `SELECT FLAT SUM(ARRAY_LENGTH(g.highestScores)) FROM Games g`

Summary

In this chapter, we learned the necessary steps for composing and executing queries against a Cosmos DB collection that uses the SQL API. Now we are able to select the most appropriate tool based on our specific needs.

We understood query results for the Cosmos DB SQL dialect. We learned that the results are JSON arrays. We used the different tools to check the resource units that Cosmos DB charges for each query and we understood its impact on the available resource units after a query is executed.

We worked with schema-agnostic queries and we used many built-in functions, including type-checking and array functions. We worked with joins, array iterations, projections, and aggregate functions. We understood how to take advantage of the Cosmos DB SQL dialect to work with a schema-agnostic document database.

Now that we have a very clear understanding of the document database and its SQL dialect, we will use the .NET Core SDK and C# to code and tune our first Cosmos DB application, which is what we are going to discuss in the next chapter.

4
Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API

In this chapter, we will use Cosmos DB, the .NET Core SDK, the SQL API, and C# to code our first Cosmos DB application. We will focus on learning about many important aspects related to the SDK in order to easily build a first version of the application. We will use dynamic objects to start quickly and we will create a baseline that we will improve in a second version.

In this chapter, we will do the following:

- Understand the requirements for the first version of an application
- Understand the main classes of the Cosmos DB SDK for .NET Core
- Create a .NET Core 2 application to interact with Cosmos DB
- Configure a Cosmos DB client
- Create or retrieve a document database
- Query and create document collections
- Retrieve a document with an asynchronous query
- Insert documents that represent competitions
- Calculate a cross-partition aggregate with an asynchronous query
- Read and update an existing document with a dynamic object
- Call asynchronous methods that create and query dynamic documents

Understanding the requirements for the first version of an application

So far, we have been working with different web-based and GUI tools to interact with the Cosmos DB service and a document database with the SQL API. We have been using a single partition and the default indexing options. Now we will leverage our existing Cosmos DB knowledge to perform the different operations we learned with the Cosmos DB .NET Core SDK. In addition, we will work with a partition key and customized indexing options.

First, we will create an application that will work with dynamic documents without any schema by taking advantage of the `dynamic` keyword to create dynamic objects. The use of this keyword is one of a few possible approaches to interacting with Cosmos DB JSON documents in .NET Core and C#. The clear advantage of this approach is that we don't need to create a class that represents the documents just to perform a few operations with a collection. We will look at many common scenarios with the .NET Core SDK and then we will create a new version of the application that will use **Plain Old CLR Objects** (**POCOs**)—that is, classes that represent the documents—and that will allow us to take full advantage of working with LINQ queries against documents. Thus, we must keep in mind that the first version of our application won't represent best practices, but it will allow us to easily dive deep into the .NET Core SDK.

We will use Visual Studio 2017 as a baseline for the examples. However, you can also run the examples in Visual Studio Code in any of its supported platforms. You can work with the Azure Cosmos DB emulator or the Cosmos DB cloud-based service. Remember that the use of cloud-based services will consume credits and charges might be billed based on the Azure subscription you have.

We will stay focused on the different tasks with a Cosmos DB database, and therefore, we will create a .NET Core 2 console application. Then, we will be able to use the improved version of this application as a baseline for other future applications that require interaction with Cosmos DB, such as a RESTful web API, an ASP.NET Core MVC web application, a mobile app, or a microservice.

We will work with a few documents that represent eSports competitions, each with a unique title. Each document will include details about the competition location, the platforms that are allowed, the games that the players will be able to play, the number of registered competitors, the competition status, and its date and time. If the competition is either in progress or finished, the document will include data about the first three positions—that is, the winners—with details about their scores and prizes.

Our first version of the application will perform the following tasks:

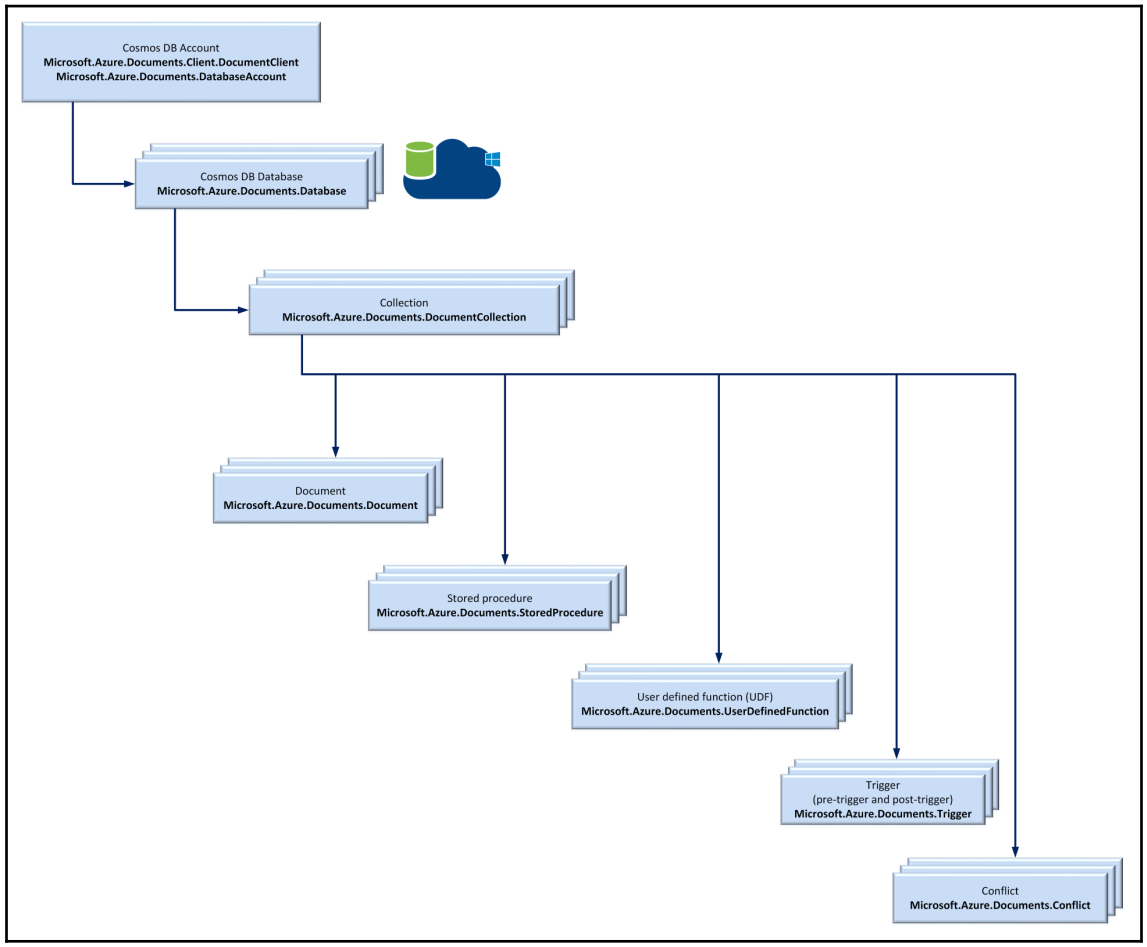
- Create a new document database if it doesn't exist
- Create a new document collection with specific options if it doesn't exist
- Retrieve a document with a specific title
- Insert a document related to a competition that has finished and has winners
- Retrieve the number of documents with a specific title
- Insert a document related to a competition that is scheduled and doesn't have winners yet
- Retrieve a document related to a competition that is scheduled and update its date and its number of registered competitors
- Retrieve and display the titles for all the scheduled competitions that have more than 200 registered competitors

Make sure you have an Azure Cosmos DB account created in Azure Portal or the Cosmos DB emulator properly installed before trying to execute any of the next examples. We will use the same account with the SQL API we have created in the previous chapters. However, remember that you can decide to use the emulator.

Understanding the main classes of the Cosmos DB SDK for .NET Core

The `Microsoft.Azure.DocumentDB.Core` NuGet package provides the client library for .NET Core to connect to Azure Cosmos DB through the SQL API. This package is essential for working with Cosmos DB in any .NET Core application. We will analyze some of the most important classes of `Microsoft.Azure.DocumentDB.Core` Version 2.0.0 before we start working on the first version of the application.

The following diagram shows the different resources that belong to a Cosmos DB account: a document database and a collection with the names of the class or classes of the Azure Cosmos DB client library, which will represent each resource below the resource name in bold. We will work with many of these classes in the first version of our application and in its next version. Note that the diagram is not a class diagram, and therefore, the arrows don't mean inheritance. The diagram shows the structure of the different resources and the classes that we will use to work with them:



The next table summarizes the information displayed in the previous diagram:

Namespace	Class name	Description
`Microsoft.Azure.Documents.Client`	`DocumentClient`	This class allows us to configure and execute requests against a specific account of the Cosmos DB service. In this case, we will only work with accounts created with the SQL API.
`Microsoft.Azure.Documents`	`DatabaseAccount`	This class represents a database account with the SQL API; that is, the container for document databases.
`Microsoft.Azure.Documents`	`Database`	This class represents a document database with the SQL API within a Cosmos DB database account.
`Microsoft.Azure.Documents`	`DocumentCollection`	This class represents a document collection (the container) within a document database.
`Microsoft.Azure.Documents`	`Document`	This class represents a JSON document that belongs to a document collection.
`Microsoft.Azure.Documents`	`StoredProcedure`	This class represents a stored procedure written with the server-side JavaScript API.
`Microsoft.Azure.Documents`	`Trigger`	This class represents a trigger written in JavaScript. The trigger can be either a pre-trigger or a post-trigger.

`Microsoft.Azure.Documents`	`Conflict`	This class represents a version conflict resource.

The following classes inherit from the `Microsoft.Azure.Documents.Resource` class:

- `DatabaseAccount`
- `Database`
- `DocumentCollection`
- `Document`
- `StoredProcedure`
- `Trigger`
- `Conflict`

The following lines show the definition for the `Microsoft.Azure.Documents.Resource` class:

```
using System;
using Newtonsoft.Json;

namespace Microsoft.Azure.Documents
{
    public abstract class Resource : JsonSerializable
    {
        protected Resource();
        protected Resource(Resource resource);

        [JsonProperty(PropertyName = "id")]
        public virtual string Id { get; set; }
        [JsonProperty(PropertyName = "_rid")]
        public virtual string ResourceId { get; set; }
        [JsonProperty(PropertyName = "_self")]
        public string SelfLink { get; }
        [JsonIgnore]
        public string AltLink { get; set; }
        [JsonConverter(typeof(UnixDateTimeConverter))]
        [JsonProperty(PropertyName = "_ts")]
        public virtual DateTime Timestamp { get; internal set; }
        [JsonProperty(PropertyName = "_etag")]
        public string ETag { get; }

        public T GetPropertyValue<T>(string propertyName);
        public void SetPropertyValue(string propertyName, object
propertyValue);
```

```
            public byte[] ToByteArray();
        }
    }
```

The use of `JsonProperty(PropertyName = attribute` followed by the name of the JSON key and a closing bracket maps a specific JSON key to the C# property, which has a different name. For example, the `_rid` JSON key will be available in the `ResourceId` property in an instance of the `Document` class. We learned about many of these properties in `Chapter 2`, *Getting Started with Cosmos DB Development and NoSQL Document Databases*, in the *Understanding the automatically generated key-value pairs* section. Specifically, we analyzed the automatically generated key-value pairs for a new document inserted in a collection. In this case, the generalized resources have fewer automatically generated values than a document. All the previously enumerated classes that represent Cosmos DB resources inherit the following standard resource properties defined in the code shown for the `Resource` class:

Resource property	**JSON key (Cosmos DB property)**	**Description**
`Id`	`id`	This string defines `id` for the resource. Notice that this `id` is different to the resource `id` that Cosmos DB uses internally. We can combine the ids for different resources to generate a URI that allows us to easily identify a resource. This URI is stored in the `AltLink` property.
`AltLink`	Not available	This string provides a unique addressable URI for the resource that combines the IDs for the different resources to generate a URI. The value of the `AltLink` property is not available as a JSON key and is automatically generated by the Cosmos DB client. We can use this property instead of the methods that allow us to build the URI with the IDs.
`ResourceId`	`_rid`	This string defines the resource ID that Cosmos DB uses internally to identify and navigate through the document resource.
`SelfLink`	`_self`	This string provides a unique addressable URI for the resource. As previously learned, we can combine self links for different resources to generate a URI that allows us to easily identify a resource. This string isn't equivalent to the previously explained `AltLink` property.
`Timestamp`	`_ts`	`Timestamp` in C# provides the last date and time at which the resource was updated.
`ETag`	`_etag`	This string provides the entity tag.

In previous versions of the Cosmos DB SDK, we were forced to use a combination of self links to identify a resource. In the newest versions, we can also combine the ID for the different resources to address a resource such as a database, a collection, or a document. We will take advantage of the possibilities included in the newest versions.

Each class that represents a resource and inherits from the `Resource` class adds its own properties. However, all of them allow us to access the previously analyzed common properties.

Creating a .NET Core 2 application to interact with Cosmos DB

Now we will create a new multiplatform .NET Core 2 console app. We will install the necessary NuGet packages to work with the Cosmos DB SDK and make it easy to use a JSON configuration file for our application.

In Visual Studio, select **File | New | Project** and select **Visual C# | .NET Core | Console App (.NET Core)**. Enter `SampleApp1` for the project name. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1.sln` file.

Install the NuGet packages and versions detailed in the next table. If there are newer versions available, you will have to verify the change log to make sure that there are no breaking changes. The sample has been tested with the specified versions:

Package name	Version
`Microsoft.Azure.DocumentDB.Core`	2.0.0
`Microsoft.Extensions.Configuration`	2.1.1
`Microsoft.Extensions.Configuration.Json`	2.1.1

As previously explained, the `Microsoft.Azure.DocumentDB.Core` package provides the client library for .NET Core to connect to Azure Cosmos DB through the SQL API. The `Microsoft.Extensions.Configuration` and `Microsoft.Extensions.Configuration.Json` packages provide a JSON key-value pair-based configuration that will allow us to easily configure the settings for the Azure Cosmos DB client.

Run the following commands within the Package Manager Console to install the previously enumerated packages:

```
Install-Package Microsoft.Azure.DocumentDB.Core -Version 2.0.0
Install-Package Microsoft.Extensions.Configuration -Version 2.1.1
Install-Package Microsoft.Extensions.Configuration.Json -Version 2.1.1
```

Now we will add a JSON configuration file for the console application with the necessary values to establish a connection with the Cosmos DB service and some additional values to specify which database and collection ids we will use to store documents. In Visual Studio, right-click on the project name (`SampleApp1`), select **Add | New Item**, and then select **Visual C# Items | Web | Scripts | JavaScript JSON Configuration File**. Enter `configuration.json` as the desired name.

Right-click on the recently added `configuration.json` file and select **Properties**. Select **Copy** if newer for the Copy to Output Directory property. This way, the configuration file will be copied to the output folder when we build the application.

Replace the contents of the new `configuration.json` file with the next lines. Make sure you replace the value specified for the `enpointUrl` key with the endpoint URL for the Azure Cosmos DB account you want to use for this example, and replace the value for the `authorizationKey` key with the read-write primary key. In this case, we will work with a new database named `Competition`, which will have a new collection named `Competitions1`. You can also make changes to these values to fulfill your requirements. The `databaseId` and `collectionId` keys define the IDs for the database and the collection that the code is going to create and use. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/configuration.json` file:

```
// Development configuration values
{
  "CosmosDB": {
    // Replace with the endpoint URL for your Azure Cosmos DB account
    "endpointUrl": "https://example001.documents.azure.com:443/",
    // Replace with the read-write primary key for your Azure Cosmos DB
account
    "authorizationKey": "Replace with the read-write primary key for your
Azure Cosmos DB account",
    // Replace with your desired database id
    "databaseId": "Competition",
    // Replace with your desired collection id
    "collectionId": "Competitions1"
  }
}
```

If you have doubts about the values you want to use, read the *Understanding URIs, read-write and read-only keys, and connection strings* section in `Chapter 2`, *Getting Started with Cosmos DB Development and NoSQL Document Databases*.

Configuring a Cosmos DB client

Now we will write the code for the main method of our console application that will retrieve the necessary values from the previously coded JSON configuration file to configure an instance of the `Microsoft.Azure.Documents.Client.DocumentClient` class, which will allow us to write additional code that will perform requests to the Azure Cosmos DB service. We will be writing methods that perform additional tasks later, and we will use many code snippets to understand each task. We will analyze many of the possible ways of performing tasks with the Cosmos DB SDK.

Replace the code of the `Program.cs` file with the following contents, which declare the necessary `using` statements, the first lines of the new `Program` class, and the `Main` static method. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
namespace SampleApp1
{
    using Microsoft.Azure.Documents;
    using Microsoft.Azure.Documents.Client;
    using Microsoft.Azure.Documents.Linq;
    using Microsoft.Extensions.Configuration;
    using System;
    using System.Linq;
    using System.Threading.Tasks;

    public class Program
    {
        private static string databaseId;
        private static string collectionId;
        private static DocumentClient client;

        public static void Main(string[] args)
        {
            var configurationBuilder = new ConfigurationBuilder();
            configurationBuilder.AddJsonFile("configuration.json",
optional: false, reloadOnChange: false);
            var configuration = configurationBuilder.Build();
            string endpointUrl = configuration["CosmosDB:endpointUrl"];
            string authorizationKey =
configuration["CosmosDB:authorizationKey"];
            databaseId = configuration["CosmosDB:databaseId"];
            collectionId = configuration["CosmosDB:collectionId"];
            try
            {
                using (client = new DocumentClient(new Uri(endpointUrl),
authorizationKey))
```

```
                {
                    CreateAndQueryDynamicDocumentsAsync().Wait();
                }
            }
            catch (DocumentClientException dce)
            {
                var baseException = dce.GetBaseException();
                Console.WriteLine(
                    $"DocumentClientException occurred. Status code:
{dce.StatusCode}; Message: {dce.Message}; Base exception message:
{baseException.Message}");
            }
            catch (Exception e)
            {
                var baseException = e.GetBaseException();
                Console.WriteLine(
                    $"Exception occurred. Message: {e.Message}; Base
exception message: {baseException.Message}");
            }
            finally
            {
                Console.WriteLine("Press any key to exit the console
application.");
                Console.ReadKey();
            }
        }
```

Notice that we will close braces after we finish coding all the methods for the `Program` class.

The `Program` class declares the following three static properties, which the different methods will use to perform tasks with the Cosmos DB service:

- `databaseId`: The Cosmos DB document database ID
- `collectionId`: The collection ID
- `client`: An instance of the `DocumentClient` class that will allow us to perform requests to the Azure Cosmos DB service

The `Main` static method creates an instance of the `Microsoft.Extensions.Configuration.ConfigurationBuilder` class named `configurationBuilder`. Then, the code calls that `configurationBuilder.AddJsonFile` method to add the previously created `configuration.json` file as a configuration provider with the key-value pairs for our work with Cosmos DB. The next line calls `configurationBuild.Build` to make all the key-value pairs available in the configuration object.

Then, the next lines retrieve the `endpointUrl`, `authorizationId`, `databaseId`, and `collectionId` values from the configuration object that grabs these values from the `configuration.json` file. The `databaseId` and `collectionId` values are stored in static fields with the same names and many methods will use them to perform operations on the document database and the collection.

Then, the code creates a new `DocumentClient` instance within a `using` statement and saves it in the `client` field. The following lines show the code that creates a URI instance with the `endpointUrl` value and passes it as a parameter with the `authorizationKey` to the `DocumentClient` constructor. The code within the `using` block calls the `CreateAndQueryDynamicDocumentsAsync` method, chained to the call to the `Wait` method, that makes it wait for the task returned by the asynchronous method to finish. Note that the `Wait` method is necessary because we have created a console application and we used the default build settings that work with C# 7.0. We could also take advantage of new features included in C# 7.1, but they would require additional steps to simplify just one line of code. We will code the `CreateAndQueryDynamicDocumentsAsync` method later, and this method will call many asynchronous methods that will have the `databaseId`, `collectionId`, and `client` fields with their appropriate values and instances to perform operations with the Cosmos DB service. Once the method finishes its execution, the resources for `DocumentClient` are cleaned up:

```
using (client = new DocumentClient(new Uri(endpointUrl), authorizationKey))
{
    CreateAndQueryDynamicDocumentsAsync().Wait();
}
```

The preceding code is enclosed in a `try...catch...finally` block that catches any `DocumentClientException` exceptions and displays the `StatusCode` and `Message` property values for this type of exception. In addition, it displays the `Message` property value for the base exception. Whenever an operation related to Cosmos DB fails, `DocumentClientException` will be thrown, which will be captured so we can see the detailed message in the console.

In addition, the code catches any exception that is not an instance of `DocumentClientException` and prints details about it and its base exception.

The code in the `finally` block asks the user to press any key to exit the console application in order to make it possible to read all the messages displayed in the console before the window is closed.

First, we will code all the static asynchronous methods that the `CreateAndQueryDynamicDocumentsAsync` static method will call and we will analyze them. Then, we will code the `CreateAndQueryDynamicDocumentsAsync` static method.

Creating or retrieving a document database

The following lines declare the code for the `RetrieveOrCreateDatabaseAsync` asynchronous static method, which creates a new document database in the Cosmos DB account if a database with `Id` equal to the value stored in the `databaseId` field doesn't exist. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<Database> RetrieveOrCreateDatabaseAsync()
{
    // Create a new document database if it doesn't exist
    var databaseResponse = await client.CreateDatabaseIfNotExistsAsync(
        new Database
        {
            Id = databaseId,
        });
    switch (databaseResponse.StatusCode)
    {
        case System.Net.HttpStatusCode.Created:
            Console.WriteLine($"The database {databaseId} has been
created.");
            break;
        case System.Net.HttpStatusCode.OK:
            Console.WriteLine($"The database {databaseId} has been
retrieved.");
            break;
    }
    return databaseResponse.Resource;
}
```

The code calls the `client.CreateDatabaseIfNotExistsAsync` asynchronous method with a new `Database` instance with its `Id` set to the `databaseId` value with no specific request options. If a document database with the specified `Id` value already exists, the method retrieves the database resource. Otherwise, the method creates a new database with the provided `Id`. Hence, the first time this method is executed, it will create a database with the provided `Id` and the default provisioning options because the code doesn't specify any values for the optional options. The second time this method is executed, it will just retrieve the existing database resource.

The `client.CreateDatabaseIfNotExistsAsync` method returns a `ResourceResponse<Database>` instance, which the code saves in the `databaseResponse` variable. The created or retrieved database resource is available in the `databaseResource.Resource` property of this instance. In this case, the `Resource` property is of the previously explained `Database` type.

The responses from the create, read, update, and delete operations on any Cosmos DB resource will return the resource response wrapped in a `ResourceResponse` instance.

The `databaseResource` instance has many properties that provide the request unit consumed by the performed activity in the `RequestCharge` property. The code checks the value of the HTTP status code available in the `databaseResponse.StatusCode` property to determine whether the database has been created or retrieved. If this property is equal to the `HTTP 201` created status (`System.Net.HttpStatusCode.Created`), it means that it was necessary to create the database because it didn't exist. If this property is equal to the `HTTP 200 OK` status (`System.Net.HttpStatusCode.OK`), it means that the database existed and it was retrieved.

Finally, the method returns the `Database` instance stored in the `databaseResponse.Resource` property.

If something goes wrong, the catch block defined in the `Main` method will capture any `DocumentClientException` or `Exception` instances and will display their details. For example, if a key is invalid, the connection won't be established with the Azure Cosmos DB service and the exception will be captured.

Querying and creating document collections

The following lines declare the code for the `CreateCollectionIfNotExistsAsync` asynchronous static method, which creates a new document collection if a collection with `id` equal to the value stored in the `collectionId` field doesn't exist in the database. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<DocumentCollection>
CreateCollectionIfNotExistsAsync()
{
    var databaseUri = UriFactory.CreateDatabaseUri(databaseId);
    DocumentCollection documentCollectionResource;
    var isCollectionCreated = await
client.CreateDocumentCollectionQuery(databaseUri)
        .Where(c => c.Id == collectionId)
        .CountAsync() == 1;
    if (isCollectionCreated)
    {
        Console.WriteLine($"The collection {collectionId} already
exists.");
        var documentCollectionUri =
UriFactory.CreateDocumentCollectionUri(databaseId, collectionId);
        var documentCollectionResponse = await
client.ReadDocumentCollectionAsync(documentCollectionUri);
        documentCollectionResource = documentCollectionResponse.Resource;
    }
    else
    {
        var documentCollection = new DocumentCollection
        {
            Id = collectionId,
        };
        documentCollection.PartitionKey.Paths.Add("/location/zipCode");
        var uniqueKey = new UniqueKey();
        uniqueKey.Paths.Add("/title");
        documentCollection.UniqueKeyPolicy.UniqueKeys.Add(uniqueKey);
        var requestOptions = new RequestOptions
        {
            OfferThroughput = 1000,
        };
        var collectionResponse = await
client.CreateDocumentCollectionAsync(
            databaseUri,
            documentCollection,
```

```
                requestOptions);
            if (collectionResponse.StatusCode ==
System.Net.HttpStatusCode.Created)
            {
                Console.WriteLine($"The collection {collectionId} has been
created.");
            }
            documentCollectionResource = collectionResponse.Resource;
        }

        return documentCollectionResource;
    }
```

The code doesn't use the easiest mechanism to create a collection when it doesn't exist because we will analyze how we can query the document collections for a database. Our goal is to learn about many possibilities offered by the SDK that will enable us to develop many different kinds of applications that work with Cosmos DB. However, it is very important to notice that the code for this method could be simplified by calling the `CreateDocumentCollectionIfNotExistsAsync` method.

First, the code calls the `UriFactory.CreateDatabaseUri` method with `databaseId` as its argument and saves the result in the `databaseUri` variable. This method will return a `Uri` instance with the URI for the database ID received as an argument. The code will use this URI to easily address the database resource in which we have to perform operations.

Note that the `UriFactory.CreateDatabaseUri` method requires the database ID to build the `Uri` instance and doesn't require any query to the database. However, of course, we must be sure that the database resource with the specified ID already exists before using the generated URI.

The next line declares the `documentCollectionResource` variable as a `DocumentCollection` instance. The code will end up returning this variable.

Then, the code creates a LINQ query with a call to the asynchronous `client.CreateDocumentCollectionQuery` method with `databaseUri` as an argument. This counts the number of collections whose `Id` is equal to `collectionId` with an asynchronous execution due to the usage of the chained `CountAsync` method. If the results of this query on the collections for the database is `1`, it means that the collection already exists.

The following lines show the code that builds the LINQ query and stores the results of the Boolean expression in the `isCollectionCreated` variable:

```
var isCollectionCreated = await
client.CreateDocumentCollectionQuery(databaseUri)
    .Where(c => c.Id == collectionId)
    .CountAsync() == 1;
```

We can easily query the existing collections in a document database by chaining LINQ expressions to the call to the `CreateDocumentCollectionQuery` method. In this case, we are always working with asynchronous methods. If we used the `Count` method instead of `CountAsync`, the query would have a synchronous execution. We will always use the asynchronous methods in the examples.

If the collection exists, the code calls the `UriFactory.CreateDocumentCollectionUri` method with `databaseId` and `collectionId` as its arguments and saves the result in the `documentCollectionUri` variable. This method will return a `Uri` instance with the URI for document collection, generated with the combination of the database ID and the collection ID received as arguments. The code will use this URI to easily address the document collection resource we want to retrieve. The next line that is executed if the collection already exists calls the `client.ReadDocumentCollectionAsync` method with `documentCollectionUri` as an argument to retrieve the collection resource with the specified URI. This method returns a `ResourceResponse<DocumentCollection>` instance that the code saves in the `documentCollectionResponse` variable. The retrieved document collection resource is available in the `documentCollectionResponse.Resource` property of this instance, which is saved in the previously declared `documentCollectionResource` variable. In this case, the `Resource` property is of the previously explained `DocumentCollection` type.

We could have retrieved the `DocumentCollection` instance with a different version of the previously explained LINQ query. However, our goal is to explore different things we can do with the Cosmos DB SDK. In fact, whenever we know the database and collection ids and we want to retrieve a `DocumentCollection` instance, the most efficient way to do so is by calling the previously explained `ReadDocumentCollectionAsync` method.

If the collection doesn't exist, the code creates a new collection within the database whose ID is equal to `databaseId`. In order to do this, it is necessary to specify the desired settings for the collections that we configured in the Azure portal with C# code. The next line in the `else` block creates a new `DocumentCollection` instance with its `Id` set to the `collectionId` value and saves it in the `documentCollection` variable. Then, the code specifies the desired partition key and the desired unique key policy for the new collection. The call to the `documentCollection.PartitionKey.Paths.Add` method with `location/zipCode` as an argument adds the `zipCode` key of the location sub-document as the desired partition key for the collection to be created. This way, our documents will be partitioned by the ZIP code in which the competition is located.

Then, the code creates a new `UniqueKey` instance, saves it in the `uniqueKey` variable, and calls the `uniqueKey.Paths.Add` method with `/title` as an argument to specify the desired unique key path to the title key. Then, the call to the `documentCollection.UniqueKeyPolicy.UniqueKeys.Add` method with `uniqueKey` as an argument adds this instance to the unique key policies for the collection to be created. This way, we will make sure that Cosmos DB won't allow us to have two competitions with the same title.

The next line creates a new `RequestOptions` instance with its `OfferThroughput` property set to `1000` and saves it in the `X` variable. This way, we indicate that we want a reserved throughput of 1,000 request units per second for the collection.

The next line calls the `client.CreateDocumentCollectionAsync` method with the following arguments:

- `databaseUri`: The `Uri` instance for the database that will hold the new collection.
- `documentCollection`: The `DocumentCollection` instance with the `id`, partition key, and unique key definitions for the collection. In this case, we are using the default indexing options. We will dive deep into other indexing options later.
- `requestOptions`: The `RequestOptions` instance with the desired reserved throughput for the collection.

The call to the `client.CreateDocumentCollectionAsync` method returns a `ResourceResponse<DocumentCollection>` instance, which the code saves in the `collectionResponse` variable. The created document collection resource is available in the `collectionResponse.Resource` property of this instance that is saved in the previously defined `documentCollectionResource` variable. In this case, the `Resource` property is of the previously explained `DocumentCollection` type.

The code checks the value of the HTTP status code available in the `collectionResponse.StatusCode` property to determine whether the collection has been created and displays a message if this property is equal to the `HTTP 201` created status (`System.Net.HttpStatusCode.Created`).

Finally, the method returns the `documentCollectionResource` variable with the `DocumentCollection` instance that was either retrieved or created.

Retrieving a document with an asynchronous query

The following lines declare the code for the `GetCompetitionByTitle` asynchronous static method, which builds a query to retrieve a competition with a specific title from the document collection. The code takes advantage of the possibilities offered by the SDK to limit the number of results returned by the query and execute the query with an asynchronous execution. The code adds complexity to work with an asynchronous execution. We don't want to run a query with a synchronous execution in our examples. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<dynamic> GetCompetitionByTitle(string
competitionTitle)
{
    // Build a query to retrieve a document with a specific title
    var collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var documentQuery = client.CreateDocumentQuery(collectionUri,
        $"SELECT * FROM Competitions c WHERE c.title =
'{competitionTitle}'",
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 1,
        })
        .AsDocumentQuery();
    while (documentQuery.HasMoreResults)
    {
        foreach (var competition in await documentQuery.ExecuteNextAsync())
        {
            Console.WriteLine(
                $"The document with the following title exists:
```

```
    {competitionTitle}");
                Console.WriteLine(competition);
                return competition;
            }
        }

        // No matching document found
        return null;
    }
```

First, the code calls the `UriFactory.CreateDocumentCollectionUri` method with `databaseId` and `collectionId` as its arguments and saves the result in the `collectionUri` variable. This method will return a `Uri` instance with the URI for the document collection. The code will use this URI to easily address the document collection resource in which we have to run a query. We must be sure that the database resource with the specified ID already exists before using the generated URI.

The next line calls the `client.CreateDocumentQuery` method to create a query with documents with the following arguments:

- `collectionUri`: The `Uri` instance for the document collection whose documents we want to query.
- `$"SELECT * FROM Competitions c WHERE c.title = '{competitionTitle}'"`: A string with the SQL API query to retrieve the competition whose title matches the string received in the `competitionTitle` parameter.
- `new FeedOptions() { EnableCrossPartitionQuery = true, MaxItemCount = 1 }`: A new `FeedOptions` instance that specifies that we want to enable the query that sends more than one request because its scope is not limited to a single partition key value. The query will check competitions whose location might have different `zipCode` values, and therefore, we assign `true` to the `EnableCrossPartitionQuery` property. In addition, we assign `1` to the `MaxItemCount` property because we want a maximum of one result each time we perform the enumeration operation by calling the asynchronous `ExecuteNextAsync` method.

The `client.CreateDocumentQuery` method returns a `System.Linq.IQueryable<dynamic>` object, which the code converts to `Microsoft.Azure.Documents.Linq.IDocumentQuery<dynamic>` by chaining a call to the `AsDocumentQuery` method. The `IDocumentQuery<dynamic>` object supports pagination and asynchronous execution and it is saved in the `documentQuery` variable. In this case, pagination is not very important because we will always have a maximum of one document that matches the criteria. Remember that we enforce a unique title value for the documents.

At this point, the query hasn't been executed. The use of the `AsDocumentQuery` method enables the code to access the `HasMoreResults` bool property in a `while` loop that makes calls to the asynchronous `ExecuteNextAsync` method to retrieve more results as long as they are available. The first time the `documentQuery.HasMoreResults` property is evaluated, its value is `true`. However, no query is executed yet. Hence, the true value indicates that we must make a call to the `documentQuery.ExecuteNextAsync` asynchronous method to retrieve the first `resultset` with a maximum number of items equal to the `MaxItemCount` value specified for the `FeedOptions` instance. After the code calls the `documentQuery.ExecuteNextAsync` method for the first time, the value of the `documentQuery.HasMoreResults` property will be updated to indicate whether another call to the `documentQuery.ExecuteNextAsync` method is necessary because another `resultset` is available.

In this case, the loop will execute the `documentQuery.ExecuteNextAsync` asynchronous method once because we expect only one item in the `resultset` if a competition with the specified title exists. However, the code uses the loop to demonstrate how the queries that retrieve documents are usually executed, and we can use the code as a baseline for other queries. In this case, we don't load pages at a different time, and therefore, we don't work with a continuation token.

A `foreach` loop iterates the `IEnumerable<dynamic>` object provided by the `FeedResponse<dynamic>` object returned by the `documentQuery.ExecuteNextAsync` asynchronous method, which enumerates the results of the appropriate page of the execution of the query. Each retrieved competition is a dynamic object that represents the document retrieved from the document collection with the query. If there is a match, the code in the `foreach` loop will display the retrieved document; that is, the competition that matches the title and the method will return this dynamic object. Otherwise, the method will return `null`.

Inserting documents that represent competitions

Now we will write two methods that insert different documents that represent competitions. First, we will code a method that works with a dynamic object to insert the following JSON document that represents the first competition that has already finished. Note that the `dateTime` value for the document will be calculated to be 50 days before today:

```
{
    "id": "1",
    "title": "Crowns for Gamers - Portland 2018",
    "location": {
        "zipCode": "90210",
        "state": "CA"
    },
    "platforms": [
        "PS4",
        "XBox",
        "Switch"
    ],
    "games": [
        "Fortnite",
        "NBA Live 19"
    ],
    "numberOfRegisteredCompetitors": 80,
    "numberOfCompetitors": 60,
    "numberOfViewers": 300,
    "status": "Finished",
    "dateTime": "2018-07-23T01:25:11.0085577Z",
    "winners": [
        {
            "player": {
                "nickName": "EnzoTheGreatest",
                "country": "Italy",
                "city": "Rome"
            },
            "position": 1,
            "score": 7500,
            "prize": 1500
        },
        {
            "player": {
                "nickName": "NicoInGamerLand",
                "country": "Argentina",
                "city": "Buenos Aires"
```

```
            },
            "position": 2,
            "score": 6500,
            "prize": 750
        },
        {
            "player": {
                "nickName": "KiwiBoy",
                "country": "New Zealand",
                "city": "Auckland"
            },
            "position": 3,
            "score": 3500,
            "prize": 250
        }
    ],
}
```

The following lines declare the code for the `InsertCompetition1` asynchronous static method, which receives the desired ID, title, and location ZIP code for the competition and inserts a new document in the document collection. Notice that, in this case, we are working with dynamic objects, and don't forget that we will create a new version of the application to use POCOs. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<Document> InsertCompetition1(string
competitionId,
    string competitionTitle,
    string competitionLocationZipCode)
{
    // Insert a document related to a competition that has finished and has
winners
    var collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var documentResponse = await client.CreateDocumentAsync(collectionUri,
new
    {
        id = competitionId,
        title = competitionTitle,
        location = new
        {
            zipCode = competitionLocationZipCode,
            state = "CA",
        },
        platforms = new[]
        {
```

```
                "PS4", "XBox", "Switch"
            },
            games = new[]
            {
                "Fortnite", "NBA Live 19"
            },
            numberOfRegisteredCompetitors = 80,
            numberOfCompetitors = 60,
            numberOfViewers = 300,
            status = "Finished",
            dateTime = DateTime.UtcNow.AddDays(-50),
            winners = new[]
            {
                new
                {
                    player = new
                    {
                        nickName = "EnzoTheGreatest",
                        country = "Italy",
                        city = "Rome"
                    },
                    position = 1,
                    score = 7500,
                    prize = 1500,
                },
                new
                {
                    player = new
                    {
                        nickName = "NicoInGamerLand",
                        country = "Argentina",
                        city = "Buenos Aires"
                    },
                    position = 2,
                    score = 6500,
                    prize = 750
                },
                new
                {
                    player = new
                    {
                        nickName = "KiwiBoy",
                        country = "New Zealand",
                        city = "Auckland"
                    },
                    position = 3,
                    score = 3500,
                    prize = 250
```

```
                    }
                },
            });

            if (documentResponse.StatusCode == System.Net.HttpStatusCode.Created)
            {
                Console.WriteLine($"The competition with the title
    {competitionTitle} has been created.");
            }

            return documentResponse.Resource;
    }
```

The first line calls the `UriFactory.CreateDocumentCollectionUri` method with `databaseId` and `collectionId` as its arguments and saves the result in the `collectionUri` variable. The next line calls the `client.CreateDocumentAsync` asynchronous method to request Cosmos DB to create a document with `collectionUri` and a new dynamic object that we want to serialize to JSON and insert as a document in the specified document collection.

The call to the `client.CreateDocumentCollectionAsync` method returns a `ResourceResponse<Document>` instance, which the code saves in the `documentResponse` variable. The created document resource is available in the `documentResponse.Resource` property of this instance. In this case, the `Resource` property is of the previously explained `Document` type.

The code checks the value of the HTTP status code available in the `documentResponse.StatusCode` property to determine whether the document has been created and displays a message if this property is equal to the `HTTP 201` created status code (`System.Net.HttpStatusCode.Created`).

Finally, the method returns the `documentResponse.Resource` variable with the `Document` instance that was created.

Now we will code a method that works with another dynamic object to insert the following JSON document, which represents a second competition that is scheduled and hasn't started yet. Note that the `dateTime` value for the document will be calculated to be 50 days from now:

```
{
    "id": "2",
    "title": "Defenders of the crown - San Diego 2018",
    "location": {
        "zipCode": "92075",
        "state": "CA"
```

```
        },
        "platforms": [
            "PC",
            "PS4",
            "XBox"
        ],
        "games": [
            "Madden NFL 19",
            "Fortnite"
        ],
        "numberOfRegisteredCompetitors": 160,
        "status": "Scheduled",
        "dateTime": "2018-10-31T01:56:49.6411125Z",
    }
```

The following lines declare the code for the `InsertCompetition2` asynchronous static method, which is very similar to the previously explained `InsertCompetition1` method. The only difference is that the dynamic object passed as an argument to the `client.CreateDocumentAsync` asynchronous method is different. Of course, we can generalize the `insert` method. However, we will do this in the second version of the application, which works with POCOs instead of using dynamic objects. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
            private static async Task<Document> InsertCompetition2(string
    competitionId,
                string competitionTitle,
                string competitionLocationZipCode)
            {
                // Insert a document related to a competition that is scheduled
                // and doesn't have winners yet
                var collectionUri =
    UriFactory.CreateDocumentCollectionUri(databaseId, collectionId);
                var documentResponse = await
    client.CreateDocumentAsync(collectionUri, new
                {
                    id = competitionId,
                    title = competitionTitle,
                    location = new
                    {
                        zipCode = competitionLocationZipCode,
                        state = "CA",
                    },
                    platforms = new[]
                    {
```

```
                        "PC", "PS4", "XBox"
                    },
                    games = new[]
                    {
                        "Madden NFL 19", "Fortnite"
                    },
                    numberOfRegisteredCompetitors = 160,
                    status = "Scheduled",
                    dateTime = DateTime.UtcNow.AddDays(50),
                });

                if (documentResponse.StatusCode ==
System.Net.HttpStatusCode.Created)
                {
                    Console.WriteLine($"The competition with the title
{competitionTitle} has been created.");
                }

                return documentResponse.Resource;
            }
```

We are working with error-prone dynamic objects. For example, if we have a typo and we write `gamess` instead of `games` in the dynamic object for the second competition, our code would generate a document that has a `gamess` key and another document that has a `games` key. Such a typo wouldn't cause a build error and we would only find out about the problem if we inspected the created documents. Hence, it is very important to understand how to work with POCOs after we finish our first version of the application.

Calculating a cross-partition aggregate with an asynchronous query

The following lines declare the code for the `DoesCompetitionWithTitleExist` asynchronous static method, which builds a query to count the number of competitions with the received title. In order to compute this aggregate, we must run a cross-partition query because the title for the competition can be at any location; that is, at any ZIP code. Cross-partition queries only support aggregates that use the `VALUE` keyword as a prefix. We learned about this keyword in the previous chapter.

As happened in other samples, there are other ways of achieving the same results. In this case, we use a similar pattern to the one we introduced in the `GetCompetitionByTitle` asynchronous static method. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<bool> DoesCompetitionWithTitleExist(string
competitionTitle)
{
    bool exists = false;
    // Retrieve the number of documents with a specific title
    // Very important: Cross partition queries only support 'VALUE
<AggreateFunc>' for aggregates
    var collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var documentCountQuery = client.CreateDocumentQuery(collectionUri,
        $"SELECT VALUE COUNT(1) FROM Competitions c WHERE c.title =
'{competitionTitle}'",
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 1,
        })
        .AsDocumentQuery();
    while (documentCountQuery.HasMoreResults)
    {
        var documentCountQueryResult = await
documentCountQuery.ExecuteNextAsync();
        exists = (documentCountQueryResult.FirstOrDefault() == 1);
    }

    return exists;
}
```

The `VALUE` keyword makes the aggregate function return only the computed value without the key. The result of the query is a single value. Hence, we don't need to use `foreach` and we can use the `FirstOrDefault` method to retrieve the count value from the `IEnumerable<dynamic>` object provided by the `FeedResponse<dynamic>` object returned by the `documentCountQuery.ExecuteNextAsync` asynchronous method, which enumerates the results of the only page of the execution of the query.

Reading and updating an existing document with a dynamic object

Now we will write a method that updates the document that represents the second scheduled competition. Specifically, the method changes the values for the `dateTime` and `numberOfRegisteredCompetitors` keys.

At the time I was writing this book, the only way to update the value of any key in a document stored in a Cosmos DB collection was to replace the document with a new one. Since the first version of Cosmos DB and the SQL API, this is the only way to update the values for keys in a document. In this case, we just need to update the values for two keys, but we will have to replace the entire document.

The following lines declare the code for the `UpdateScheduledCompetition` asynchronous static method, which receives the competition ID, its location ZIP code, the new date and time, and the new number of registered competitors in the `competitionId`, `competitionLocationZipCode`, `newDataTime`, and `newNumberOfRegisteredCompetitors` arguments. The method retrieves the document whose ID matches the `competitionId` value received as an argument, casts the retrieved document as a dynamic object to update the values for the explained keys, and uses this dynamic object to replace the existing document. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task<Document> UpdateScheduledCompetition(string
competitionId,
    string competitionLocationZipCode,
    DateTime newDateTime,
    int newNumberOfRegisteredCompetitors)
{
    // Retrieve a document related to a competition that is scheduled
    // and update its date and its number of registered competitors
    // The read operation requires the partition key
    var documentToUpdateUri = UriFactory.CreateDocumentUri(databaseId,
collectionId, competitionId);
    var readDocumentResponse = await
client.ReadDocumentAsync(documentToUpdateUri, new RequestOptions()
    {
        PartitionKey = new PartitionKey(competitionLocationZipCode)
    });
    ((dynamic)readDocumentResponse.Resource).dateTime = newDateTime;
```

```
        ((dynamic)readDocumentResponse.Resource).numberOfRegisteredCompetitors
    = newNumberOfRegisteredCompetitors;
        ResourceResponse<Document> updatedDocumentResponse = await
    client.ReplaceDocumentAsync(
            documentToUpdateUri,
            readDocumentResponse.Resource);

        if (updatedDocumentResponse.StatusCode == System.Net.HttpStatusCode.OK)
        {
            Console.WriteLine($"The competition with id {competitionId} has
    been updated.");
        }

        return updatedDocumentResponse.Resource;
    }
```

The first line calls the `UriFactory.CreateDocumentUri` method with `databaseId`, `collectionId`, and `competitionId` (the document ID) as their arguments and saves the result in the `documentToUpdateUri` variable. This method will return a `Uri` instance with the URI for the document, generated with the combination of the database ID, the collection `id`, and the document ID received as arguments. The code will use this URI to easily address the document resource we want to retrieve.

The next line calls the `client.ReadDocumentAsync` asynchronous method with `documentToUpdateUri` and a new `RequestOptions` instance as the arguments. The value for the `PartitionKey` property of the `RequestOptions` instance is initialized to a new `PartitionKey` instance with the received `competitionLocationZipCode` received as an argument. This way, we specify the URI and the partition key to enable us to retrieve the document in the simplest and cheapest read operation when working with a partitioned document collection.

The `client.ReadDocumentAsync` method returns a `ResourceResponse<Document>` instance, which the code saves in the `readDocumentResponse` variable. The retrieved document resource is available in the `readDocumentResponse.Resource` property of this instance. In this case, the `Resource` property is of the previously explained `Document` type.

The next two lines cast the `Document` instance in the `readDocumentResponse.Resource` property to dynamic to set the new value for the `dateTime` and `numberOfRegisteredCompetitors` properties. It is necessary to cast to a dynamic object because we aren't using POCOs to represent the documents.

The next line calls the `client.ReplaceDocumentAsync` method with the following arguments:

- `documentToUpdateUri`: The `Uri` instance for the document that will be replaced
- `readDocumentResponse.Resource`: The document with the new values for the `dateTime` and `numberOfRegisteredCompetitors` properties

The use of dynamic objects might cause issues the appropriate type returned by the `client.ReplaceDocumentAsync` asynchronous method. Hence, in this case, the code specifies the type for the `updatedDocumentResponse` variable to which we assign the results of the asynchronous call.

The call to the `client.ReplaceDocumentAsync` method returns a `ResourceResponse<Document>` instance, which the code saves in the `updatedDocumentResponse` variable. The created document resource is available in the `updatedDocumentResponse.Resource` property of this instance. In this case, the `Resource` property is of the previously explained `Document` type.

The code checks the value of the HTTP status code available in the `collectionResponse.StatusCode` property to determine whether the document has been updated and displays a message if this property is equal to the `HTTP 200 OK` status (`System.Net.HttpStatusCode.OK`).

Finally, the method returns the `updatedDocumentResponse.Resource` property with the `Document` instance that was updated.

Querying documents in multiple partitions

The following lines declare the code for the `ListScheduledCompetitions` asynchronous static method, which builds a query to retrieve the titles for all the scheduled competitions that have more than 200 registered competitors and shows them in the console output. In order to retrieve these titles, we must run a cross-partition query because the competitions with more than 200 registered competitors can be at any location; that is, at any ZIP code.

As happened in other samples, there are other ways of achieving the same results. In this case, we use a similar pattern to the one we introduced in the `GetCompetitionByTitle` asynchronous static method. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
private static async Task ListScheduledCompetitions()
{
    // Retrieve the titles for all the scheduled competitions that have
more than 200 registered competitors
    var collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var selectTitleQuery = client.CreateDocumentQuery(collectionUri,
        $"SELECT VALUE c.title FROM Competitions c WHERE
c.numberOfRegisteredCompetitors > 200 AND c.status = 'Scheduled'",
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 100,
        })
        .AsDocumentQuery();
    while (selectTitleQuery.HasMoreResults)
    {
        var selectTitleQueryResult = await
selectTitleQuery.ExecuteNextAsync();
        foreach (var title in selectTitleQueryResult)
        {
            Console.WriteLine(title);
        }
    }
}
```

In this case, the `FeedOptions` instance specifies that we want a `MaxItemCount` of 100 documents per page—that is, per call to the `ExecuteNextAsync` method—to iterate through the `resultset` page. We don't filter any specific `zipCode`, and therefore, we set the `EnableCrossPartitionQuery` to `true` to enable a cross-partition query.

The use of the `VALUE` keyword in the query makes it possible to retrieve the values without a key, and we can easily write a `foreach` block that writes a line in the console for each retrieved title.

Calling asynchronous methods that create and query dynamic documents

Now we will write the code for the `CreateAndQueryDynamicDocumentsAsync` asynchronous static method, which calls the previously created and explained asynchronous static methods. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_04_01` folder in the `dot_net_core_2_samples/SampleApp1/SampleApp1/Program.cs` file:

```
        private static async Task CreateAndQueryDynamicDocumentsAsync()
        {
            var database = await RetrieveOrCreateDatabaseAsync();
            Console.WriteLine(
                $"The database {databaseId} is available for operations
with the following AltLink: {database.AltLink}");
            var collection = await CreateCollectionIfNotExistsAsync();
            Console.WriteLine(
                $"The collection {collectionId} is available for operations
with the following AltLink: {collection.AltLink}");
            string competition1Id = "1";
            string competition1Title = "Crowns for Gamers - Portland 2018";
            string competition1ZipCode = "90210";
            var competition1 = await
GetCompetitionByTitle(competition1Title);
            if (competition1 == null)
            {
                competition1 = await InsertCompetition1(competition1Id,
competition1Title, competition1ZipCode);
            }

            string competition2Title = "Defenders of the crown - San Diego
2018";
            bool isCompetition2Inserted = await
DoesCompetitionWithTitleExist(competition2Title);
            string competition2Id = "2";
            string competition2LocationZipCode = "92075";
            if (isCompetition2Inserted)
            {
                Console.WriteLine(
                    $"The document with the following title exists:
{competition2Title}");
            }
            else
            {
```

```
                var competition2 = await InsertCompetition2(competition2Id,
competition2Title, competition2LocationZipCode);
            }

            var updatedCompetition2 = await
UpdateScheduledCompetition(competition2Id,
                competition2LocationZipCode,
                DateTime.UtcNow.AddDays(60),
                250);

            await ListScheduledCompetitions();
        }
    }
}
```

The new method that is called by the `Main` method performs the following actions:

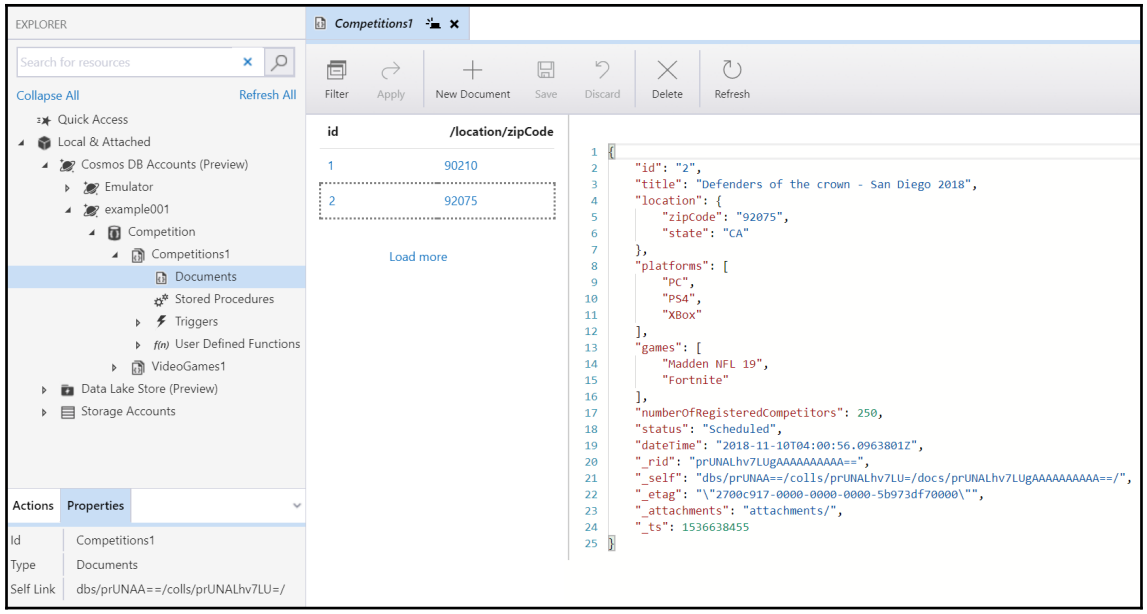
1. Calls the `RetrieveOrCreateDatabaseAsync` method to create or retrieve the Cosmos DB document database specified in the appropriate key in the `configuration.json` file.
2. Calls the `CreateCollectionIfNotExistsAsync` method to create or retrieve the Cosmos DB document collection specified in the appropriate key in the `configuration.json` file.
3. Calls the `GetCompetitionByTitle` method to check whether a competition with a title that matches `Crowns for Gamers - Portland 2018` exists. If the competition isn't found, the code calls the `InsertCompetition1` method to insert the document that represents the first competition.
4. Calls the `DoesCompetitionWithTitleExist` method to check whether a competition with a title that matches `Defenders of the crown - San Diego 2018` exists. If the competition isn't found, the code calls the `InsertCompetition2` method to insert the document that represents the second competition.
5. Calls the `UpdateScheduledCompetition` method to update the date and time and the registered number of competitions for the second competition.
6. Calls the `ListScheduledCompetitions` method to list the titles for all the scheduled competitions that have more than 200 registered competitors.

Now run the application for the first time and you will see the following messages in the console output:

```
The database Competition has been retrieved.
The database Competition is available for operations with the following
AltLink: dbs/Competition
```

```
The collection Competitions1 has been created.
The collection Competitions1 is available for operations with the following
AltLink: dbs/Competition/colls/Competitions1
The competition with the title Crowns for Gamers - Portland 2018 has been
created.
The competition with the title Defenders of the crown - San Diego 2018 has
been created.
The competition with id 2 has been updated.
Defenders of the crown - San Diego 2018
Press any key to exit the console application.
```

Use your favorite tool to check the documents in the Cosmos DB database and collection that you have configured in the `configuration.json` file that the application uses. Make sure you refresh the appropriate screen in the selected tool. You will see two documents that belong to different partitions based on the value of the `location.zipCode` key. For example, the following screenshot shows the inserted and updated document whose `id` is equal to `2` and its `location.zipCode` is equal to `92075` in Microsoft Azure Storage Explorer:



Now run the application for the second time and you will see the following messages similar to the following in the console output:

```
The database Competition has been retrieved.
The database Competition is available for operations with the following
```

```
AltLink: dbs/Competition
The collection Competitions1 already exists.
The collection Competitions1 is available for operations with the following
AltLink: dbs/Competition/colls/Competitions1
The document with the following title exists: Crowns for Gamers - Portland
2018
{"id":"1","title":"Crowns for Gamers - Portland
2018","location":{"zipCode":"90210","state":"CA"},"platforms":["PS4","XBox"
,"Switch"],"games":["Fortnite","NBA Live
19"],"numberOfRegisteredCompetitors":80,"numberOfCompetitors":60,"numberOfV
iewers":300,"status":"Finished","dateTime":"2018-07-23T04:00:52.2062076Z","
winners":[{"player":{"nickName":"EnzoTheGreatest","country":"Italy","city":
"Rome"},"position":1,"score":7500,"prize":1500},{"player":{"nickName":"Nico
InGamerLand","country":"Argentina","city":"Buenos
Aires"},"position":2,"score":6500,"prize":750},{"player":{"nickName":"KiwiB
oy","country":"New
Zealand","city":"Auckland"},"position":3,"score":3500,"prize":250}],"_rid":
"prUNALhv7LUfAAAAAAAAAA==","_self":"dbs/prUNAA==/colls/prUNALhv7LU=/docs/pr
UNALhv7LUfAAAAAAAAAA==/","_etag":"\"2700c717-0000-0000-0000-5b973df30000\""
,"_attachments":"attachments/","_ts":1536638451}
The document with the following title exists: Defenders of the crown - San
Diego 2018
The competition with id 2 has been updated.
Defenders of the crown - San Diego 2018
Press any key to exit the console application.
```

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. Which of the following classes represents a document collection:
    1. `Microsoft.Azure.Documents.Collection`
    2. `Microsoft.Azure.Documents.DocumentCollection`
    3. `Microsoft.Azure.Documents.Client.Collection`
2. Which of the following classes represents a JSON document that belongs to a document collection:
    1. `Microsoft.Azure.Documents.Document`
    2. `Microsoft.Azure.Documents.Collection.Document`
    3. `Microsoft.Azure.Documents.Client.Document`

3. In the `Microsoft.Azure.Documents.Resource` class, which property is mapped to the JSON key ID:
    1. `ResourceId`
    2. `Id`
    3. `SelfLink`
4. The responses from create, read, update and delete operations on any Cosmos DB resource return the resource response wrapped in an instance of which of the following classes:
    1. `Resource`
    2. `Response`
    3. `ResourceResponse`
5. Which of the following classes provides static methods to create URI instances for databases, document collections, documents and other resources:
    1. `UriFactory`
    2. `UriBuilder`
    3. `Uri`

Summary

In this chapter, we worked with the main classes of the Cosmos DB SDK for .NET Core and we built our first .NET Core 2 application that interacts with Cosmos DB. We configured the Cosmos DB client and we wrote code to create or retrieve a document database, query and create document collections, and retrieve documents with asynchronous queries.

We wrote code that used dynamic objects to insert documents that represented competitions. We read and updated existing documents with dynamic objects and we calculated cross-partition aggregates.

Now that we have a very clear understanding of the basics of the .NET Core SDK with dynamic objects to perform create, read, and update operations with Cosmos DB, we will work with POCOs and LINQ queries, which are the topics we are going to discuss in the next chapter.

5
Working with POCOs, LINQ, and a NoSQL Document Database

In this chapter, we will continue working with the .NET Core SDK, but this time we will work with POCOs and LINQ queries. We will take advantage of the strongly typed features of C# and the functional programming features that LINQ provides to work with Cosmos DB. We will improve the application we started in the previous chapter and we will understand the advantages of working with POCOs combined with LINQ.

In this chapter, we will do the following:

- Create models and customize serialization
- Insert POCOs
- Calculate a cross-partition aggregate with an asynchronous LINQ query
- Read and update an existing document with a POCO
- Query documents in multiple partitions with LINQ
- Write LINQ queries that perform operations on arrays
- Call asynchronous methods that use POCOs to create and query documents
- Inspect the SQL API queries that LINQ generates

Creating models and customizing serialization

So far, we have been working with dynamic objects and we wrote SQL queries in strings without taking advantage of the beloved LINQ features. Now we will create a new version of the application that will use POCOs to represent the competitions. This way, we will be able to use strongly typed properties and work with LINQ to build queries instead of composing queries with strings.

Whenever we have to persist a document in the document database, the C# object that represents the document will be serialized to a JSON document; that is, it will be encoded in a string. Whenever we have to retrieve a document from the document database, the JSON document will be deserialized to the C# object that represents the document; that is, the object will be built from the string.

One of the key benefits of working with Cosmos DB, its .NET Core SDK, and a document database based on the SQL API is that we don't have to use an **Object-Relational Mapping** (**ORM**) as we usually would when working with relational databases and object-oriented programming languages such as C#. With Cosmos DB, we just need to provide the necessary instructions to the JSON serializer for the classes that represent the documents whenever the default options aren't suitable. There is no need to perform any additional configuration tasks. This is much simpler than working with ORMs.

The fact that we decide to work with POCOs doesn't mean that we will lose the benefits of the schema-agnostic document database. In fact, we will write the code for the first version of the application and then we will make changes based on new requirements that will generate different schemas, and so we won't require any migration process.

Now we will use the .NET Core 2 console application we coded in the previous chapter as a baseline and we will code the following new elements:

- `CompetitionStatus`: This `enum` defines the three possible statuses of a competition: scheduled, finished, and canceled. We will use the appropriate serialization settings to serialize the `enum` value as a string instead of as a number. Hence, for example, `CompetitionStatus.Finished` will be serialized as `"Finished"`.
- `Location`: This class defines a competition's location.
- `Player`: This class defines a player.

- `Winner`: This class defines a competition's winner.
- `Competition`: This class defines a competition with a location, and its winners if its status is `"Finished"`. This class represents the document we will persist in the document collection.

The code file for the solution with the new sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1.sln` file.

In Visual Studio, right-click on the project name (**SampleApp1**) in **Solution Explorer**, select **Add | New Folder**, and enter `Types` as the desired name.

Right-click on the **Types** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `CompetitionStatus.cs` as the desired name.

Enter the following code to declare `CompetitionStatusenum` in the `CompetitionStatus.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Types/CompetitionStatus.cs` file:

```
namespace SampleApp1.Types
{
    public enum CompetitionStatus
    {
        // The competition is scheduled and didn't happen yet
        Scheduled,
        // The competition is finished and has winners
        Finished,
        // The competition has been canceled
        Canceled
    }
}
```

Right-click on the **Types** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `GamingPlatform.cs` as the desired name.

Enter the following code to declare the `GamingPlatformenum` in the `GamingPlatform.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Types/GamingPlatform.cs` file:

```
namespace SampleApp1.Types
{
    using Newtonsoft.Json;
    using Newtonsoft.Json.Converters;

    [JsonConverter(typeof(StringEnumConverter))]
```

```
        public enum GamingPlatform
        {
            Switch,
            PC,
            PS4,
            XBox,
            iOS,
            Android
        }
    }
```

The `enum` declaration is preceded by the following attribute, which overrides the default JSON converter to indicate that we want to serialize the `enum` declaration to its string representation. For example, `GamingPlatform.PS4` will be serialized to `"PS4"`:

```
[JsonConverter(typeof(StringEnumConverter))]
```

In Visual Studio, right-click on the project name (**SampleApp1**) in **Solution Explorer**, select **Add | New Folder**, and enter `Models` as the desired name.

Right-click on the **Models** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `Location.cs` as the desired name.

Enter the following code to declare the `Location` class in the `Location.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Models/Location.cs` file:

```
namespace SampleApp1.Models
{
    using Newtonsoft.Json;

    public class Location
    {
        [JsonProperty(PropertyName = "zipCode")]
        public string ZipCode { get; set; }

        [JsonProperty(PropertyName = "state")]
        public string State { get; set; }
    }
}
```

In C#, public properties use upper CamelCase, also known as PascalCase, and therefore, start with an uppercase character. In our previous example, we used lower camelCase for the keys in the JSON documents, and therefore, the keys in the two documents inserted in the document collection start with a lowercase letter.

The `ZipCode` property for an instance of the `Location` class must be serialized to a key-value pair whose key must be `"zipCode"` instead of `"ZipCode"`. Cosmos DB uses Json.NET, a popular JSON framework for .NET, to serialize POCOs to JSON documents and deserialize JSON documents to POCOs. The default behavior of Json.NET makes it convert each property name in the C# class to a JSON key without performing any case changes. Thus, if we use the default settings, `ZipCode` will be serialized to a key named `ZipCode`. JSON is case sensitive, and therefore, `zipCode` is not the same as `ZipCode` for a key.

We might solve this issue by configuring a contract resolver to handle this case conversion automatically in the serialization and deserialization processes handled by Json.NET. However, we want to stay focused and we will keep the example simple, so we will use attributes for each property that requires a different name. Just keep in mind there are simpler ways when we want to work with more classes.

For example, if we run the following query in the `Competitions1` document collection, the results will be an array of empty JSON objects because the `ZipCode` key doesn't exist within the object in the `location` key of the two existing documents:

```
SELECT c.location.ZipCode FROM c
```

The following lines show the results of the previous query:

```
[
    {},
    {}
]
```

The next query uses the appropriate case for the `zipCode` key:

```
SELECT c.location.zipCode FROM c
```

The following lines show the results of the new query with the appropriate case:

```
[
    {
        "zipCode": "90210"
    },
    {
        "zipCode": "92075"
    }
]
```

We can use the `Newtonsoft.Json.JsonProperty` attribute with the `PropertyName` parameter set to the name of the JSON key to which a property must be mapped. In the previous code, the `ZipCode` property is mapped to the `"zipCode"` JSON key and the `State` property is mapped to the `"state"` key. We will use the `JsonProperty` attribute for all the properties we define for our models to indicate the appropriate name for the JSON key that is mapped to each property.

In this case, we are creating models that must be compatible with existing documents that have been inserted in the document collection. However, in other cases, we might decide to use the same case for the C# properties and the JSON document keys to avoid using the `JsonProperty` attribute for each property.

Right-click on the **Models** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `Player.cs` as the desired name.

Enter the following code to declare the `Player` class in the `Player.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Models/Player.cs` file:

```
namespace SampleApp1.Models
{
    using Newtonsoft.Json;

    public class Player
    {
        [JsonProperty(PropertyName = "nickName")]
        public string NickName { get; set; }

        [JsonProperty(PropertyName = "country")]
        public string Country { get; set; }

        [JsonProperty(PropertyName = "city")]
        public string City { get; set; }
    }
}
```

Right-click on the **Models** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `Winner.cs` as the desired name.

Enter the following code to declare the `Winner` class in the `Winner.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Models/Winner.cs` file:

```
namespace SampleApp1.Models
{
    using Newtonsoft.Json;

    public class Winner
    {
        [JsonProperty(PropertyName = "player")]
        public Player Player { get; set; }

        [JsonProperty(PropertyName = "position")]
        public int Position { get; set; }

        [JsonProperty(PropertyName = "score")]
        public int Score { get; set; }

        [JsonProperty(PropertyName = "prize")]
        public int Prize { get; set; }
    }
}
```

Right-click on the **Models** folder, select **Add | New Item**, and then select **Visual C# Items | Code | Code File**. Enter `Competition.cs` as the desired name.

Enter the following code to declare the `Competition` class in the `Competition.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Models/Competition.cs` file:

```
namespace SampleApp1.Models
{
    using Microsoft.Azure.Documents;
    using Newtonsoft.Json;
    using Newtonsoft.Json.Converters;
    using SampleApp1.Types;
    using System;
    public class Competition: Document
    {
        [JsonProperty(PropertyName = "title")]
        public string Title { get; set; }

        [JsonProperty(PropertyName = "location")]
        public Location Location { get; set; }

        [JsonProperty(PropertyName = "platforms")]
```

```
        public GamingPlatform[] Platforms { get; set; }

        [JsonProperty(PropertyName = "games")]
        public string[] Games { get; set; }

        [JsonProperty(PropertyName = "numberOfRegisteredCompetitors")]
        public int NumberOfRegisteredCompetitors { get; set; }

        [JsonProperty(PropertyName = "numberOfCompetitors")]
        public int NumberOfCompetitors { get; set; }

        [JsonProperty(PropertyName = "numberOfViewers")]
        public int NumberOfViewers { get; set; }

        [JsonProperty(PropertyName = "status")]
        [JsonConverter(typeof(StringEnumConverter))]
        public CompetitionStatus Status { get; set; }

        [JsonProperty(PropertyName = "dateTime")]
        public DateTime DateTime { get; set; }

        [JsonProperty(PropertyName = "winners")]
        public Winner[] Winners { get; set; }
    }
}
```

The `Competition` class represents the document that is persisted in the document collection and inherits from the `Microsoft.Azure.Documents.Document` class. This way, a `Competition` instance will be able to access the automatically generated key-value pairs through the properties inherited from the superclass of `Document`: `Resource`. We learned about these properties in the *Understanding the main classes of the Cosmos DB SDK for .NET Core* section in `Chapter 4`, *Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API*.

The `Competition` class declares the `Status` property as an `enum` declaration of the previously created `CompetitionStatus` type. The following attribute overrides the default JSON converter to indicate that we want to serialize the `enum` to its string representation:

```
[JsonConverter(typeof(StringEnumConverter))]
```

We have coded the necessary models that will enable us to write code that inserts `Competition` instances in the Cosmos DB document collection, retrieve them, and write strongly typed LINQ queries.

Retrieving a POCO with a LINQ asynchronous query

Now open the `Program.cs` file and add the following `using` statements after the last line that declares a using statement: `using System.Threading.Tasks;`. The code file for the sample is included in the `learning_cosmos_db_06_01` folder in the `SampleApp1/SampleApp1/Program.cs` file:

```
using System.Collections.Generic;
using SampleApp1.Models;
using SampleApp1.Types;
```

Add the following static field to the `Program` class after the code that declares the `client` static field: `private static DocumentClient client;`. In this new version, we will save the `Uri` instance for the document collection in this field to reuse it in all the methods that require the collection URI:

```
private static Uri collectionUri;
```

The following lines declare the code for the `GetCompetitionByTitleWithLinq` asynchronous static method that builds a LINQ query to retrieve a `Competition` instance with a specific title from the document collection. The code is the LINQ version of the existing `GetCompetitionByTitle` static method. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task<Competition> GetCompetitionByTitleWithLinq(string
title)
{
    // Build a query to retrieve a Competition with a specific title
    var documentQuery =
client.CreateDocumentQuery<Competition>(collectionUri,
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 1,
        })
        .Where(c => c.Title == title)
        .Select(c => c)
        .AsDocumentQuery();
    while (documentQuery.HasMoreResults)
    {
```

```
        foreach (var competition in await
documentQuery.ExecuteNextAsync<Competition>())
        {
            Console.WriteLine(
                $"The Competition with the following title exists:
{title}");
            Console.WriteLine(competition);
            return competition;
        }
    }

    // No matching document found
    return null;
}
```

First, the code calls the `client.CreateDocumentQuery<Competition>` method to create a query for documents of the `Competition` type with the following arguments:

- `collectionUri`: The `Uri` instance for the document collection whose documents we want to query.
- `new FeedOptions() { EnableCrossPartitionQuery = true, MaxItemCount = 1 }`: A new `FeedOptions` instance that specifies that we want to enable the query that sends more than one request, because its scope is not limited to a single partition key value. The query will check `Competition` instances whose `Location` property might have different `ZipCode` values, and therefore, we assign `true` to the `EnableCrossPartitionQuery` property. In addition, we assign `1` to the `MaxItemCount` property because we want a maximum of `1` result each time we perform the enumeration operation.

The `client.CreateDocumentQuery<Competition>` method returns a `System.Linq.IOrderedQueryable<Competition>` object, which the code converts to `Microsoft.Azure.Documents.Linq.IDocumentQuery<Competition>` by chaining a call to the `AsDocumentQuery` method after the chained LINQ query methods. The `IDocumentQuery<Competition>` object supports pagination and asynchronous execution and it is saved in the `documentQuery` variable.

The `Where` query method checks whether the `Title` property matches the title received as an argument, and the `Select` query method indicates we want to retrieve the `Competition` instance for each `Competition` that matches the criteria.

At this point, the query hasn't been executed. The use of the `AsDocumentQuery` method enables the code to access the `HasMoreResults` bool property in a `while` loop that makes calls to the asynchronous `ExecuteNextAsync` method to retrieve more results as long as they are available. The first time the `documentQuery.HasMoreResults` property is evaluated, its value is `true`. However, no query was executed yet. Hence, the `true` value indicates that we must make a call to the `documentQuery.ExecuteNextAsync<Competition>` asynchronous method to retrieve the first result set with a maximum number of items equal to the `MaxItemCount` value specified for the `FeedOptions` instance. After the code calls the `documentQuery.ExecuteNextAsync<Competition>` method for the first time, the value of the `documentQuery.HasMoreResults` property will be updated to indicate whether another call to the `documentQuery.ExecuteNextAsync<Competition>` method is necessary because another result set is available.

Note that we specify the `type` argument for the `documentQuery.ExecuteNextAsync` method within angle brackets (`<Competition>`) to make sure that the `competition` variable will be of the `Competition` type that we retrieve with the query. If we didn't specify the `type` argument, and we used `var competition` to declare the variable in the `foreach` loop, C# would use the dynamic type for competition, and we don't want this to happen.

A `foreach` loop iterates the `IEnumerable<Competition>` object provided by the `FeedResponse<Competition>` object returned by the `documentQuery.ExecuteNextAsync<Competition>` asynchronous method, which enumerates the results of the appropriate page of the execution of the query. Each retrieved `competition` is a `Competition` instance that Json.NET has deserialized from the JSON document retrieved from the document collection with the LINQ query. In addition, the LINQ query generates a SQL API query for the Cosmos DB document collection.

If there is a match, the code in the `foreach` loop will display the retrieved document; that is, the `Competition` instance that matches the title and the method will return this `Competition` instance. Otherwise, the method will return `null`. Note that the method returns `Task<Competition>`, and therefore, the code that calls this method will be able to work with a `Competition` instance instead of dealing with a dynamic object.

Note that the previous method doesn't chain a `FirstOrDefault` method to the query, because this method would execute the query with a synchronous execution. In our examples, we are always working with queries with an asynchronous execution and we won't use synchronous methods to run the queries.

Inserting POCOs

Now we will write the generalized `InsertCompetition` asynchronous method, which receives a `Competition` instance, inserts it into the document collection, and displays a message indicating that the competition with a specific status and title has been created. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task<Competition> InsertCompetition(Competition
competition)
{
    var documentResponse = await client.CreateDocumentAsync(collectionUri,
competition);

    if (documentResponse.StatusCode == System.Net.HttpStatusCode.Created)
    {
        Console.WriteLine($"The {competition.Status} competition with the
title {competition.Title} has been created.");
    }

    Competition insertedCompetition = (dynamic) documentResponse.Resource;
    return insertedCompetition;
}
```

The first line calls the `client.CreateDocumentAsync` asynchronous method to request that Cosmos DB creates a document with `collectionUri` and the received `Competition` instance, which we want to serialize to JSON and insert as a document in the specified document collection.

The call to the `client.CreateDocumentCollectionAsync` method returns a `ResourceResponse<Document>` instance, which the code saves in the `documentResponse` variable. The created document resource is available in the `documentResponse.Resource` property of this instance.

The code checks the value of the HTTP status code available in the `documentResponse.StatusCode` property to determine whether the document has been created, and displays a message if this property is equal to the `HTTP 201` created status (`System.Net.HttpStatusCode.Created`).

Then, the code casts the `documentResponse.Resource` variable to a `dynamic` object and uses it to generate the `Competition` instance, which was created as a document in the collection, and saves it in the `insertedCompetition` variable. Finally, the method returns this `Competition` instance.

Now we will write three methods that insert new documents that represent competitions by working with instances of the `Competition` class and its related classes. The methods will end up calling the recently coded `InsertCompetition` method.

The following lines declare the code for the `InsertCompetition3` asynchronous static method, which creates and inserts a finished competition with two winners. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task<Competition> InsertCompetition3()
{
    var competition = new Competition
    {
        Id = "3",
        Title = "League of legends - San Diego 2018",
        Location = new Location
        {
            ZipCode = "92075",
            State = "CA"
        },
        Platforms = new[]
        {
            GamingPlatform.Switch
        },
        Games = new[]
        {
            "Fortnite", "NBA Live 19", "PES 2019"
        },
        NumberOfRegisteredCompetitors = 80,
        NumberOfCompetitors = 30,
        NumberOfViewers = 390,
        Status = CompetitionStatus.Finished,
        DateTime = DateTime.UtcNow.AddDays(-20),
        Winners = new[]
```

```
            {
                new Winner
                {
                    Player = new Player
                    {
                        NickName = "BrandonMilazzo",
                        Country = "Italy",
                        City = "Milazzo"
                    },
                    Position = 1,
                    Score = 12850,
                    Prize = 800
                },
                new Winner
                {
                    Player = new Player
                    {
                        NickName = "Kevin Maverick",
                        Country = "Ireland",
                        City = "Dublin"
                    },
                    Position = 2,
                    Score = 12500,
                    Prize = 400
                },
            },
        };

        return await InsertCompetition(competition);
    }
```

The following lines declare the code for the `InsertCompetition4` asynchronous static method, which creates and inserts a scheduled competition. Note that the `dateTime` value for the document will be calculated to be 300 days from now. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
    private static async Task<Competition> InsertCompetition4()
    {
        // Insert a document related to a competition that is scheduled
        // and doesn't have winners yet
        var competition = new Competition
        {
            Id = "4",
            Title = "League of legends - San Diego 2019",
            Location = new Location
            {
```

```
                ZipCode = "92075",
                State = "CA"
            },
            Platforms = new[]
            {
                GamingPlatform.Switch, GamingPlatform.PC, GamingPlatform.XBox
            },
            Games = new[]
            {
                "Madden NFL 19", "Fortnite"
            },
            Status = CompetitionStatus.Scheduled,
            DateTime = DateTime.UtcNow.AddDays(300),
        };

        return await InsertCompetition(competition);
    }
```

The fact that we are working with strongly typed code to create the competitions ensures that we don't have typos in the property names. In addition, the use of enums that are serialized to strings makes sure that we don't have typos for the competition status and the supported gaming platforms.

Calculating a cross-partition aggregate with an asynchronous LINQ query

The following lines declare the code for the `DoesCompetitionWithTitleExistWithLinq` asynchronous static method, which builds a LINQ query to count the number of competitions with the received title. In order to compute this aggregate, we must run a cross-partition query, because the title for the competition can be at any location; that is, at any zip code.

Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task<bool>
DoesCompetitionWithTitleExistWithLinq(string competitionTitle)
{
    var competitionsCount = await
client.CreateDocumentQuery<Competition>(collectionUri,
        new FeedOptions()
```

```
            {
                EnableCrossPartitionQuery = true,
                MaxItemCount = 1,
            })
            .Where(c => c.Title == competitionTitle)
            .CountAsync();

        return (competitionsCount == 1);
    }
```

First, the code calls the `client.CreateDocumentQuery<Competition>` method to create a query for documents of the `Competition` type with the following arguments:

- `collectionUri`: The `Uri` instance for the document collection whose documents we want to query
- `new FeedOptions() { EnableCrossPartitionQuery = true, MaxItemCount = 1 }`: A new `FeedOptions` instance that specifies that we want to enable cross-partition queries

The `client.CreateDocumentQuery<Competition>` method returns a `System.Linq.IOrderedQueryable<Competition>` object, which the code chains to a `Where` LINQ query method and to the `CountAsync` asynchronous method to retrieve the number of competitions that match the criteria with an asynchronous query. The `CountAsync` asynchronous method performs the `COUNT` aggregate function for the competitions that match the specified criteria with the `Where` LINQ query method.

The `CountAsync` asynchronous method is declared in the `Microsoft.Azure.Documents.Linq` namespace. This namespace also provides the following asynchronous methods that we can use to calculate aggregates for the different numeric types:

- `AverageAsync`: This method computes an average
- `MaxAsync`: This method computes the maximum value
- `MinAsync`: This method computes the minimum value
- `SumAsync`: This method computes the sum of all the values

Reading and updating an existing document with a POCO

Now we will write a method that updates a persisted `Competition` instance that represents a competition that is scheduled. Specifically, the method changes the values for the `DateTime`, `NumberOfRegisteredCompetitors`, and `Platforms` properties.

The following lines declare the code for the `UpdateScheduledCompetitionWithPlatforms` asynchronous static method, which receives the competition `id`, its location zip code, the new date and time, the new number of registered competitors, and the new gaming platforms in the `competitionId`, `competitionLocationZipCode`, `newDataTime`, `newNumberOfRegisteredCompetitors`, and `newGamingPlatforms` arguments. The method retrieves the `Competition` instance whose ID matches the `competitionId` value received as an argument, and updates the values to the explained properties. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task<Competition>
UpdateScheduledCompetitionWithPlatforms(string competitionId,
    string competitionLocationZipCode,
    DateTime newDateTime,
    int newNumberOfRegisteredCompetitors,
    IList<GamingPlatform> newGamingPlatforms)
{
    // Retrieve a document related to a competition that is scheduled
    // and update its date, number of registered competitors and platforms
    // The read operation requires the partition key
    var documentToUpdateUri = UriFactory.CreateDocumentUri(databaseId,
collectionId, competitionId);
    var readCompetitionResponse = await
client.ReadDocumentAsync<Competition>(documentToUpdateUri, new
RequestOptions()
    {
        PartitionKey = new PartitionKey(competitionLocationZipCode)
    });
    readCompetitionResponse.Document.DateTime = newDateTime;
    readCompetitionResponse.Document.NumberOfRegisteredCompetitors =
newNumberOfRegisteredCompetitors;
    readCompetitionResponse.Document.Platforms =
newGamingPlatforms.ToArray();
    var updatedCompetitionResponse = await client.ReplaceDocumentAsync(
        documentToUpdateUri,
```

```
            readCompetitionResponse.Document);

        if (updatedCompetitionResponse.StatusCode ==
    System.Net.HttpStatusCode.OK)
        {
            Console.WriteLine($"The competition with id {competitionId} has
    been updated.");
        }

        Competition updatedCompetition =
    (dynamic)updatedCompetitionResponse.Resource;
        return updatedCompetition;
    }
```

The call to the `client.ReadDocumentAsync<Competition>` asynchronous method with `documentToUpdateUri` and a new `RequestOptions` instance as the arguments returns a `DocumentResponse<Competition>` instance, which the code saves in the `readCompetitionResponse` variable. The retrieved `Competition` instance deserialized from the read document is available in the `readCompetitionResponse.Document` property of this instance. Note that we specify the `type` argument for the `client.ReadDocumentAsync` method within angle brackets (`<Competition>`).

The next two lines assign the new desired values to the retrieved `Competition` instance. Note that we don't need to cast the `Document` property because it is already of the `Competition` type.

The next line calls the `client.ReplaceDocumentAsync` method with the following arguments:

- `documentToUpdateUri`: The `Uri` instance for the document that will be replaced
- `readCompetitionResponse.Document`: The `Competition` instance with the new values for the `DateTime`, `NumberOfRegisteredCompetitors`, and `Platforms` properties

The call to the `client.ReplaceDocumentAsync` method returns a `ResourceResponse<Document>` instance, which the code saves in the `updatedCompetitionResponse` variable. The persisted `Competition` instance is available in the `updatedCompetitionResponse.Resource` property of this instance. In this case, the `Resource` property is of the `Document` type.

The code checks the value of the HTTP status code available in the `collectionResponse.StatusCode` property to determine whether the document has been updated and displays a message if this property is equal to the HTTP 200 OK status (`System.Net.HttpStatusCode.OK`).

Then, the code casts the `updatedDocumentResponse.Resource` variable to a `dynamic` object and uses it to generate the `Competition` instance that was updated as a document in the collection and save it in the `updatedCompetition` variable. Finally, the method returns this `Competition` instance.

Querying documents in multiple partitions with LINQ

The following lines declare the code for the `ListScheduledCompetitionsWithLinq` asynchronous static method, which builds a LINQ query to retrieve the titles for all the scheduled competitions that have more than five registered competitors and show them in the console output.

Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task ListScheduledCompetitionsWithLinq()
{
    // Retrieve the titles for all the scheduled competitions that have
more than 5 registered competitors
    var selectTitleQuery =
client.CreateDocumentQuery<Competition>(collectionUri,
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 100,
        })
        .Where(c => (c.NumberOfRegisteredCompetitors > 5)
        && (c.Status == CompetitionStatus.Scheduled))
        .Select(c => c.Title)
        .AsDocumentQuery();
```

```
        while (selectTitleQuery.HasMoreResults)
        {
            var selectTitleQueryResult = await
    selectTitleQuery.ExecuteNextAsync<string>();
            foreach (var title in selectTitleQueryResult)
            {
                Console.WriteLine(title);
            }
        }
    }
```

The `client.CreateDocumentQuery<Competition>` method returns a `System.Linq.IOrderedQueryable<Competition>` object, which the code converts to `Microsoft.Azure.Documents.Linq.IDocumentQuery<Competition>` by chaining a call to the `AsDocumentQuery` method after the chained LINQ query methods.

The `Where` query method checks whether the `NumberOfRegisteredCompetitors` property is greater than 5 and whether the value of the `Status` property is equal to `CompetitionStatus.Scheduled`. The `Select` query method indicates we want to retrieve the `Title` property for each `Competition` that matches the specified criteria. Note that writing the criteria is easier than writing a SQL string, which was an error-prone practice.

Note that we specify the type argument for the `documentQuery.ExecuteNextAsync` method within angle brackets (`<string>`) to make sure that the `title` variable will be of the `string` type and not a dynamic object.

A `foreach` loop iterates the `IEnumerable<string>` object provided by the `FeedResponse<string>` object returned by the `documentQuery.ExecuteNextAsync<string>` asynchronous method, which enumerates the results of the appropriate page of the execution of the query. Each retrieved `title` is a `Title` property value for a `Competition` instance that Json.NET has deserialized from the `"title"` key of the JSON document retrieved from the document collection with the LINQ query.

Writing LINQ queries that perform operations on arrays

The following lines declare the code for the `ListFinishedCompetitionsFirstWinner` asynchronous static method. This method builds a LINQ query to retrieve the winner with the first position for all the finished competitions. The competitions are filtered to include only those that allowed the platform received as an argument and that are located in the zip code received as a parameter. The query restricts the location's zip code value, and therefore, it is a single-partition query.

Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task
ListFinishedCompetitionsFirstWinner(GamingPlatform gamingPlatform, string
zipCode)
{
    // Retrieve the winner with the first position for all the finished
competitions
    // that allowed the platform received as an argument
    // and located in the zipCode received as an argument.
    var winnersQuery =
client.CreateDocumentQuery<Competition>(collectionUri,
        new FeedOptions()
        {
            MaxItemCount = 100,
        })
        .Where(c => (c.Location.ZipCode == zipCode)
        && (c.Status == CompetitionStatus.Finished)
        && (c.Platforms.Contains(gamingPlatform)))
        .Select(c => c.Winners[0])
        .AsDocumentQuery();

    while (winnersQuery.HasMoreResults)
    {
        var winnersQueryResult = await
winnersQuery.ExecuteNextAsync<Winner>();
        foreach (var winner in winnersQueryResult)
        {
            Console.WriteLine($"Nickname: {winner.Player.NickName}, Score:
{winner.Score}");
        }
    }
}
```

The `Where` query method uses the `c.Platforms.Contains` method to check whether the `Platforms` array of `GamingPlatform` contains the `GamingPlatform` received in the `gamingPlatform` argument. The SQL API query generated by LINQ will use the previously learned `ARRAY_CONTAINS` built-in array function.

The `Select` query method retrieves the first winner in the `Winners` array by specifying `c.Winners[0]`. All the finished competitions have at least one winner, and therefore, we didn't need to check the number of elements in the array that would add complexity to the query.

Note that we specify the type argument for the `documentQuery.ExecuteNextAsync` method within angle brackets (`<Winner>`) to make sure that the `winner` variable will be of the `Winner` type and not a dynamic object. The `foreach` loop displays the `NickName` for the `Player` object related to a `Winner` and the achieved `Score`.

Calling asynchronous methods that use POCOs to create and query documents

Now we will write the code for the `CreateAndQueryCompetitionsWithLinqAsync` asynchronous static method, which calls the previously created and explained asynchronous static methods. Add the following lines to the existing code of the `Program.cs` file. The code file for the sample is included in the `learning_cosmos_db_05_01` folder in the `SampleApp2/SampleApp1/Program.cs` file:

```
private static async Task CreateAndQueryCompetitionsWithLinqAsync()
{
    var database = await RetrieveOrCreateDatabaseAsync();
    Console.WriteLine(
        $"The database {databaseId} is available for operations with the
following AltLink: {database.AltLink}");
    var collection = await CreateCollectionIfNotExistsAsync();
    Console.WriteLine(
        $"The collection {collectionId} is available for operations with
the following AltLink: {collection.AltLink}");
    collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var competition3 = await GetCompetitionByTitleWithLinq("League of
legends - San Diego 2018");
    if (competition3 == null)
    {
        competition3 = await InsertCompetition3();
    }
```

```
    bool isCompetition4Inserted = await
DoesCompetitionWithTitleExistWithLinq("League of legends - San Diego
2019");
    Competition competition4;
    if (isCompetition4Inserted)
    {
        competition4 = await GetCompetitionByTitleWithLinq("League of
legends - San Diego 2018");
        Console.WriteLine(
            $"The {competition4.Status} competition  with the following
title exists: {competition4.Title}");
    }
    else
    {
        competition4 = await InsertCompetition4();
    }

    var updatedCompetition4 = await
UpdateScheduledCompetitionWithPlatforms("4",
        "92075",
        DateTime.UtcNow.AddDays(300),
        10,
        new List<GamingPlatform>
        {
            GamingPlatform.PC, GamingPlatform.XBox
        });

    await ListScheduledCompetitionsWithLinq();
    await ListFinishedCompetitionsFirstWinner(GamingPlatform.PS4, "90210");
    await ListFinishedCompetitionsFirstWinner(GamingPlatform.Switch,
"92075");
}
```

Now find the following line in the existing `Main` method:

```
CreateAndQueryDynamicDocumentsAsync().Wait();
```

Replace the line with the next line:

```
CreateAndQueryCompetitionsWithLinqAsync().Wait();
```

The new method that is called by the `Main` method performs the following actions:

1. Calls the `RetrieveOrCreateDatabaseAsync` method to create or retrieve the Cosmos DB document database specified in the appropriate key in the `configuration.json` file.
2. Calls the `CreateCollectionIfNotExistsAsync` method to create or retrieve the Cosmos DB document collection specified in the appropriate key in the `configuration.json` file.
3. Calls the `GetCompetitionByTitleWithLinq` method to check whether a competition with the title that matches `"League of legends - San Diego 2018"` exists. If the competition isn't found, the code calls the `InsertCompetition3` method to insert the `Competition` instance that represents the third competition.
4. Calls the `DoesCompetitionWithTitleExistWithLinq` method to check whether a competition with the title that matches `"League of legends - San Diego 2019"` exists. If the competition isn't found, the code calls the `InsertCompetition4` method to insert the `Competition` instance that represents the fourth competition.
5. Calls the `UpdateScheduledCompetitionWithPlatforms` method to update the date and time, the registered number of competitions, and the gaming platforms for the fourth competition.
6. Calls the `ListScheduledCompetitionsWithLinq` method to list the titles for all the scheduled competitions that have more than five registered competitors.
7. Calls the `ListFinishedCompetitionsFirstWinner` method to list the nickname and achieved score for the first position winner of all the finished competitions that allowed the PS4 platform and had a zip code of 90210.
8. Calls the `ListFinishedCompetitionsFirstWinner` method to list the nickname and achieved score for the first position winner of all the finished competitions that allowed the Switch platform and had a zip code of 92075.

Now run the application for the first time and you will see the following messages in the console output:

```
    The database Competition has been retrieved.
    The database Competition is available for operations with the following
AltLink: dbs/Competition
    The collection Competitions1 already exists.
    The collection Competitions1 is available for operations with the
following AltLink: dbs/Competition/colls/Competitions1
    The Finished competition with the title League of legends - San Diego
2018 has been created.
    The Scheduled competition with the title League of legends - San Diego
2019 has been created.
    The competition with id 4 has been updated.
    Defenders of the crown - San Diego 2018
    League of legends - San Diego 2019
    Nickname: EnzoTheGreatest, Score: 7500
    Nickname: BrandonMilazzo, Score: 12850
```

Use your favorite tool to check the documents in the Cosmos DB database and collection that you have configured in the `configuration.json` file that the application uses. Make sure you refresh the appropriate screen in the selected tool. Now you will see four documents that belong to two different partitions based on the value of the `location.zipCode` key.

Inspecting the SQL API queries that LINQ generates

Now we will establish a breakpoint in one of the methods that builds a LINQ query against the Cosmos DB database to inspect the SQL API query that LINQ generates. This way, we will be able to grab the SQL queries and run them in our favorite tool to query the document collection. In addition, we will learn how to add the necessary code to print the generated SQL query in the debug output whenever necessary.

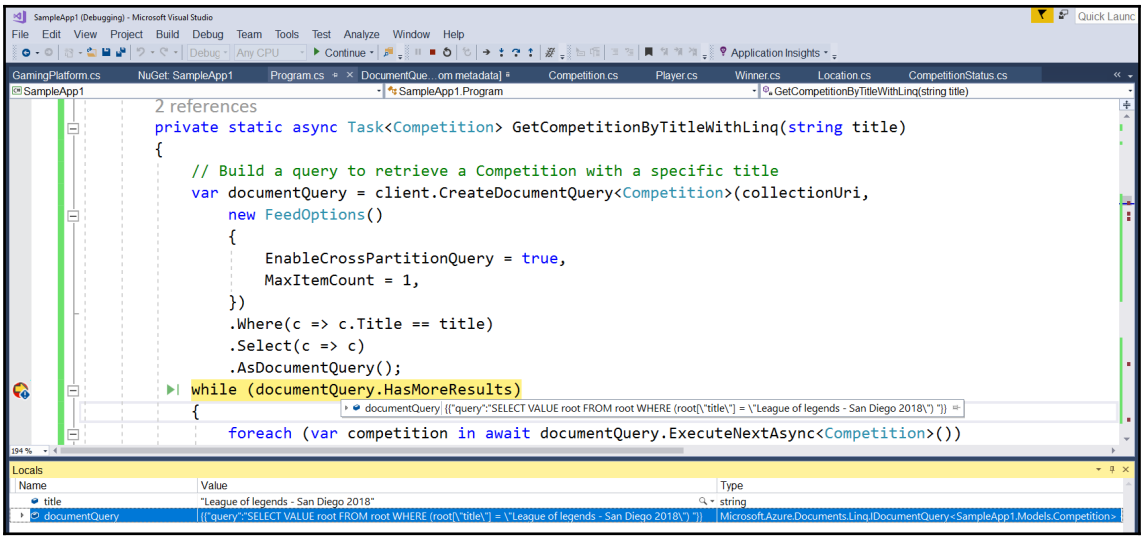
Go to the following line within the `GetCompetitionByTitleWithLinq` static method:

```
while (documentQuery.HasMoreResults)
```

Right-click on the line and select **Breakpoint | Insert breakpoint** in the context menu.

Start debugging the application.

Inspect the value for `documentQuery` and Visual Studio will display a JSON key-value pair with the generated SQL API query string in the value for the `"query"` key, as shown in the following screenshot:



We can add the following line to display the query key and its value to the debug output:

```
System.Diagnostics.Debug.WriteLine(documentQuery.ToString());
```

We can also write the following line in the **Immediate** window:

```
documentQuery.ToString()
```

The following line shows the results:

```
{"query":"SELECT VALUE root FROM root WHERE (root[\"title\"] = \"League of
legends - San Diego 2018\") "}
```

The following lines show the value for the query key with the generated SQL API query:

```
SELECT VALUE root FROM root WHERE (root[\"title\"] = \"League of legends -
San Diego 2018\")
```

Note that LINQ uses `root` as the alias name followed by the property name enclosed in square brackets, backslashes, and double quotes to access each key, instead of using the alias followed by a dot and the key name. For example, LINQ generates the following expression to evaluate whether a title is equal to `"League of legends – San Diego 2018"`:

```
root[\"title\"] = \"League of legends – San Diego 2018\"
```

The previous line is equivalent to the following expression:

```
root.title = "League of legends – San Diego 2018"
```

We can copy the query value and paste it in our favorite tool to check the results.

Test your knowledge

1. Which of the following methods is an asynchronous method declared in the `Microsoft.Azure.Documents.Linq` namespace that allows us to execute a query with an aggregate function to compute the total number of elements:
    1. `AsCountQuery`
    2. `Count`
    3. `CountAsync`
2. Which of the following attributes specify Json.Net that we want to serialize an enum to its string representation:
    1. `[JsonConverter(typeof(StringEnumConverter))]`
    2. `[JsonConverter(typeof(String))]`
    3. `[JsonConverter(typeof(EnumToString))]`
3. Which of the following methods executes a query to a Cosmos DB document collection with an asynchronous execution and returns a result set based on the `MaxItemCount` value specified in the `FeedOptions` instance:
    1. `ExecuteNextAsync`
    2. `AsDocumentQuery`
    3. `QueryAsync`
4. Which of the following attributes specify Json.Net that we want to map a property to the `prize` key in the JSON document:
    1. `[JsonProperty(Name = "prize")]`
    2. `[JsonProperty(PropertyName = "prize")]`
    3. `[JsonProperty(KeyName = "prize")]`

5. Which of the following methods reads a document with an asynchronous execution:
    1. `ReadDocument`
    2. `ReadDocumentAsync`
    3. `ReadNextDocumentAsync`

Summary

In this chapter, we learned how to take advantage of strongly typed code in C#, LINQ, and Cosmos DB. First, we created models and other necessary types, and we customized the default serialization provided by Json.NET to match our specific needs.

We worked with the main classes of the Cosmos DB SDK for .NET Core and we built a second version of a .NET Core 2 application, which interacts with Cosmos DB by using POCOs instead of dynamic objects. We wrote code to retrieve instances of a specific class with asynchronous LINQ queries.

We wrote code that persisted strongly typed objects into documents that represent competitions. We read and updated existing documents with POCOs and we composed and executed single-partition and cross-partition queries with LINQ.

Now that we have a very clear understanding of how to use of POCOs and LINQ with the .NET Core SDK to perform create, read, and update operations with Cosmos DB, we will learn how to optimize the resource units consumed, work with different indexing options, and monitor an application that uses Cosmos DB, which are the topics we are going to discuss in the next chapter.

6
Tuning and Managing Scalability with Cosmos DB

In this chapter, we will analyze many aspects that allow us to design and maintain scalable architectures with Cosmos DB. We will use our sample application to understand how many important things work, but we will also work with other examples to understand complex topics related to scalability.

In this chapter, we will do the following:

- Understand request units and how they affect billing
- Dynamically adjust throughput for a collection with the Azure portal
- Work with client-side throughput management
- Understand rate limiting and throttling
- Track consumed request units with client-side code
- Understand the options for provisioning request units
- Learn partitioning strategies
- Deploy to multiple regions
- Understand the five consistency levels
- Take advantage of regional failover
- Understand indexing in Cosmos DB
- Check indexing policies for a collection with the Azure portal

Understanding request units and how they affect billing

In `Chapter 3`, *Writing and Running Queries on NoSQL Document Databases*, we learned how to execute SQL queries against a document collection with the SQL API by using different tools. We also learned how to check the request units consumed by each query. In `Chapter 4`, *Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API*, we performed operations and composed queries in strings and we executed them against a document collection with the Cosmos DB .NET Core SDK. In `Chapter 5`, *Working with POCOs, LINQ, and a NoSQL Document Database*, we performed operations with POCOs and we composed queries with LINQ and POCOs. However, in these last two chapters, we didn't check request unit consumption.

Every operation and query performed on Cosmos DB consumes request units. We can consider request units as the main currency for Cosmos DB services.

Now we will learn what request units are, how they affect billing, and how we can measure the request unit consumption for different operations and queries with the Cosmos DB .NET Core SDK.

If you want to run a NoSQL database server in your development computer, the service will use your existing hardware resources; that is, some memory, some CPU cores, and some storage. Depending on the operations performed, the database server will consume an amount of memory, use a percentage of the available CPU cores, and perform read and write operations to the storage resources.

In order to perform operations and run queries in Cosmos DB, you just need request units. Request units combine all the resources required to execute any operation or query into a single currency. You don't need to worry about how much memory, CPU time, or storage throughput a particular query might require. You just need to pay attention to a cost expressed in request units that combine all these factors.

Request units are also known as RU in the Azure portal and many Cosmos DB tools.

Request units are the currency used for purchasing resources in Cosmos DB, and we use them as a unit to provision resources in Cosmos DB for our containers or databases. In our previous examples, we provisioned resources for a specific collection; that is, for a container of a document database.

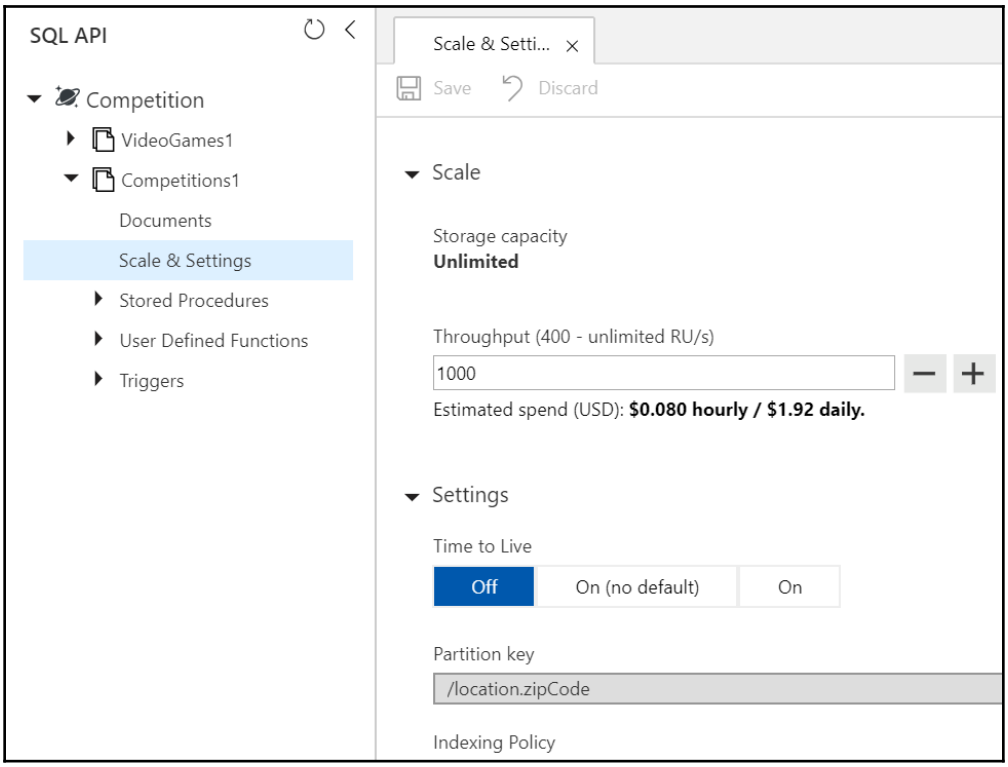
Cosmos DB provisions request units on a per-second basis, and therefore, we will see instances of **RU/s**, which means request units per second, in the Azure portal and many of the Cosmos DB tools. Azure bills the provisioned request units on a per-hour basis. Thus, whenever we configure the request units per second we want to provision for a container in the Azure portal, the portal displays the estimated hourly and daily spend in the currency that Azure has configured.

Dynamically adjusting throughput for a collection with the Azure portal

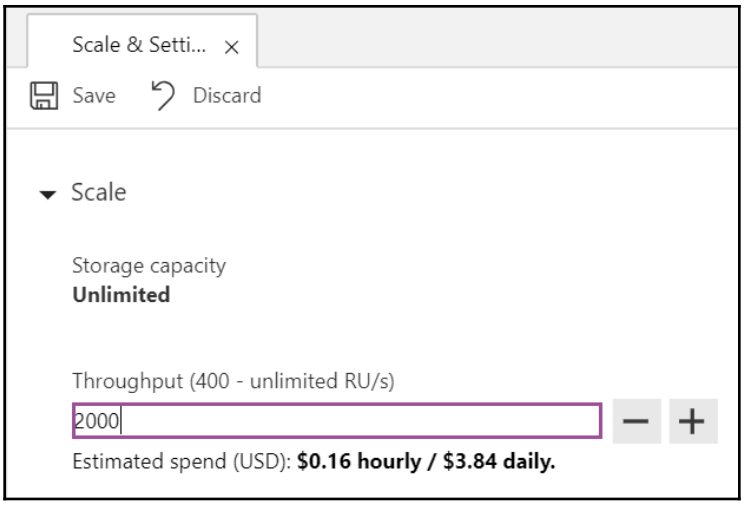
Now we will use the Azure portal to check the settings we used to create the `Competitions1` document collection with the C# code we wrote in `Chapter 4`, *Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API*.

In the Azure portal, make sure you are in the page for the Cosmos DB account in the portal. Click on the **Data Explorer** option, click on the database name you used in the configuration for the examples in `Chapter 4`, *Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API*, and `Chapter 5`, *Working with POCOs, LINQ, and a NoSQL Document Database*, (`Competition`) to expand the collections for the database and click on the collection name you used for the examples (`Competitions1`). Click on **Scale & Settings** and the portal will display a new tab that allows us to check the existing provisioned throughput for the collection in the **Scale** section. The following screenshot shows the storage capacity set to **Unlimited**, the throughput set to `1000` RU/s, and the hourly and daily estimated spend expressed in **USD** for the provisioned storage and throughput we have defined.

Note that the values might be different based on your account settings. The `CreateCollectionIfNotExistsAsync` method created the document collection with the following settings:



Replace `1000` with `2000` in the **Throughput (400 - unlimited RU/s)** textbox. Note that the estimated hourly and daily spend increases in a proportional way. In the sample configuration, which might differ from yours, the hourly estimated spend expressed in USD increases from **0.080** to **0.16** USD. The following screenshot shows how the portal previews the estimated spend for a new throughput value:



Note that we need to specify values equal or greater than 1,000 RU/s to be able to work with unlimited storage capacity because we are working with a partitioned document collection. Keep in mind that any partitioned container in Cosmos DB needs to have a minimum throughput of 1,000 RU/s to enable unlimited storage.

Once we have calculated the RU/s throughput that we need for the tasks performed by our application and for our scalability needs, we might have specific days (perhaps weekends or other certain times) where we expect more operations than usual. For example, let's consider our eSports competition sample application. Some days, the number of competitions might be higher than on other days. In fact, Fridays and Saturdays have higher number of eSports competitions than other days of the week.

As previously learned, it is very clear that an excess in the provisioned throughput will mean we will be billed for throughput that we don't need. Hence, it's useful to know that it is extremely easy to make changes to the provisioned throughput with the Azure portal, the other tools we have analyzed, and the SDKs. In addition, keep in mind that we can automate this task through scripting, as happens with many Azure maintenance tasks. However, the automation options are outside of the scope of this book.

Now we will provision more throughput for the `Competitions1` collection. Just click the **Save** button on the toolbar on top of the **Scale and Settings** tab. After a few seconds, you will see a message similar to the following one in the **Notifications** tab at the bottom of the page:

```
Successfully updated offer for resource dbs/prUNAA==/colls/prUNAJOG6yw=/
```

Once the eSports competitions application's peak demand is finished, we can follow the same steps to go back to the previously provisioned throughput of 1,000 RU/s to avoid being excessively billed for throughput we don't need.

Working with client-side throughput management

One of the nice things about Cosmos DB is that all the operations that the SDK exposes are ones that we can perform with the Cosmos DB service. Now we will use the .NET Core 2 console application that we coded in the previous chapter as a baseline, and we will make changes to our existing application to adjust the provisioned throughput for the existing `Comptetitions1` collection with C# code that uses the Cosmos DB .NET Core SDK.

Specifically, we will increase the provisioned throughput to 2,000 RU/s before we start performing the different operations and running the queries, and then, we will reduce the provisioned throughput to 1,000 RU/s before the application finishes its execution. In this way, we will understand how to dynamically scale the provisioned throughput.

Note that each different available Cosmos DB SDK ends up calling a REST API to interact with the Cosmos DB service. However, we are always working with C# and the Cosmos DB .NET Core SDK in this book.

The code file for the solution with the new sample is included in the `learning_cosmos_db_06_01` folder in the `SampleApp3/SampleApp1.sln` file.

Now open the `Program.cs` file and add the following `UpdateOfferForCollectionAsync` static method to the existing `Program` class. The code file for the sample is included in the `learning_cosmos_db_06_01` folder in the `SampleApp3/SampleApp1/Program.cs` file:

```
private static async Task<Offer> UpdateOfferForCollectionAsync(string
collectionSelfLink, int newOfferThroughput)
{
    // Create an asynchronous query to retrieve the current offer for the
document collection
    // Notice that the current version of the API only allows to use the
SelfLink for the collection
    // to retrieve its associated offer
    Offer existingOffer = null;
    var offerQuery = client.CreateOfferQuery()
        .Where(o => o.ResourceLink == collectionSelfLink)
```

```
            .AsDocumentQuery();
        while (offerQuery.HasMoreResults)
        {
            foreach (var offer in await offerQuery.ExecuteNextAsync<Offer>())
            {
                existingOffer = offer;
            }
        }
        if (existingOffer == null)
        {
            throw new Exception("I couldn't retrieve the offer for the
    collection.");
        }
        // Set the desired throughput to newOfferThroughput RU/s for the new
    offer built based on the current offer
        var newOffer = new OfferV2(existingOffer, newOfferThroughput);
        var replaceOfferResponse = await client.ReplaceOfferAsync(newOffer);

        return replaceOfferResponse.Resource;
    }
```

In a nutshell, the code retrieves the `Microsoft.Azure.Documents.Offer` instance, which represents the offer bound to the document collection with an asynchronous query. Then, the code creates a new instance of the `Microsoft.Azure.Documents.OfferV2` class, which is a subclass of `Offer`, and sets its desired provisioned throughput to the RU/s received in the `newOfferThroughput` argument. Finally, the code replaces the offer related to the document collection with the new instance of the `OfferV2` class in an asynchronous operation and returns the persisted instance. This way, we have a new method that allows us to set the desired throughput for our partitioned document collection.

The method receives the value of the `SelfLink` property for a `Collection` instance in the `collectionSelfLink` parameter, because the version of the API that we are using doesn't allow us to use a URI to retrieve the offer related to a collection. Hence, we must work with the old-fashioned self link, which was more common for other operations in the first versions of the API.

The first line declares the `existingOffer` variable as an `Offer` instance. Then, the code creates a LINQ query with a call to the `client.CreateOfferQuery` method without arguments chained to a `Where` query method, which specifies that we want to retrieve the `Offer` instance whose `ResourceLink` property matches the `collectionSelfLink` value received as an argument. This way, we indicate that we want to retrieve the offer that is related to the collection.

The `client.CreateOfferQuery` method returns a `System.Linq.IOrderedQueryable<Offer>` object, which the code converts to `Microsoft.Azure.Documents.Linq.IDocumentQuery<Offer>` by chaining a call to the `AsDocumentQuery` method after the chained LINQ query method. The `IDocumentQuery<Offer>` object supports pagination and asynchronous execution and it is saved in the `offerQuery` variable.

At this point, the query hasn't been executed. The usage of the `AsDocumentQuery` method enables the code to access the `HasMoreResults` bool property in a `while` loop that makes calls to the asynchronous `ExecuteNextAsync<Offer>` method to retrieve more results as long as they are available. In this case, we will only retrieve one `Offer` instance that matches the specified criteria. If there is a match, the code in the `foreach` loop will assign the retrieved `Offer` instance to the `existingOffer` variable.

Note that the previous method doesn't chain a `FirstOrDefault` method to the query, because this method would execute the query with a synchronous execution. In our examples, we are always working with queries with an asynchronous execution and we won't use synchronous methods to run the queries.

In our case, we will always be able to retrieve an offer related to the collection, and therefore, `existingOffer` will always have a value after the execution of the `foreach` block and performs the previously explained code to create the new `OfferV2` instance, set the new desired throughput, and persist it with an asynchronous operation.

Now replace the code for the existing `CreateAndQueryCompetitionsWithLinqAsync` static method in the `Program` class with the following lines. Note that the added lines are highlighted in the next code snippet. The code file for the sample is included in the `learning_cosmos_db_06_01` folder in the `SampleApp3/SampleApp1/Program.cs` file:

```
private static async Task CreateAndQueryCompetitionsWithLinqAsync()
{
    var database = await RetrieveOrCreateDatabaseAsync();
    Console.WriteLine(
        $"The database {databaseId} is available for operations with the
following AltLink: {database.AltLink}");
    var collection = await CreateCollectionIfNotExistsAsync();
    Console.WriteLine(
        $"The collection {collectionId} is available for operations with
the following AltLink: {collection.AltLink}");
```

```
    // Increase the provisioned throughput for the collection to 2000 RU/s
    var offer1 = await UpdateOfferForCollectionAsync(collection.SelfLink,
2000);
    Console.WriteLine(
        $"The collection {collectionId} has been re-configured with a
provision throuhgput of 2000 RU/s");
    collectionUri = UriFactory.CreateDocumentCollectionUri(databaseId,
collectionId);
    var competition3 = await GetCompetitionByTitleWithLinq("League of
legends - San Diego 2018");
    if (competition3 == null)
    {
        competition3 = await InsertCompetition3();
    }

    bool isCompetition4Inserted = await
DoesCompetitionWithTitleExistWithLinq("League of legends - San Diego
2019");
    Competition competition4;
    if (isCompetition4Inserted)
    {
        competition4 = await GetCompetitionByTitleWithLinq("League of
legends - San Diego 2019");
        Console.WriteLine(
            $"The {competition4.Status} competition  with the following
title exists: {competition4.Title}");
    }
    else
    {
        competition4 = await InsertCompetition4();
    }

    var updatedCompetition4 = await
UpdateScheduledCompetitionWithPlatforms("4",
        "92075",
        DateTime.UtcNow.AddDays(300),
        10,
        new List<GamingPlatform>
        {
            GamingPlatform.PC, GamingPlatform.XBox
        });

    await ListScheduledCompetitionsWithLinq();
    await ListFinishedCompetitionsFirstWinner(GamingPlatform.PS4, "90210");
    await ListFinishedCompetitionsFirstWinner(GamingPlatform.Switch,
"92075");
    // Decrease the provisioned throughput for the collection to 1000 RU/s
    var offer2 = await UpdateOfferForCollectionAsync(collection.SelfLink,
```

```
1000);
    Console.WriteLine(
        $"The collection {collectionId} has been re-configured with a
provision throuhgput of 1000 RU/s");
}
```

The new lines added at the beginning of the code call the previously defined `UpdateOfferForCollectionAsync` asynchronous method with the `collection.SelfLink` value and `2000` as the values for the `collectionSelfLink` and `newOfferThroughput` arguments. This way, the call provides the value of the `SelfLink` property for the previously retrieved collection and increases the provisioned throughput for the collection to 2,000 RU/s.

The new lines added at the end of the code call the previously defined `UpdateOfferForCollectionAsync` asynchronous method with the `collection.SelfLink` value and `1000` as the values for the `collectionSelfLink` and `newOfferThroughput` arguments. This way, the code decreases the provisioned throughput for the collection to 1,000 RU/s.

We can easily make changes to the provisioned throughput based on our specific needs with a few lines of code and a generic method that does the job, such as the `UpdateOfferForCollectionAsync` static method.

Understanding rate limiting and throttling

If we hit the provisioned RU/s rate limit for any operation or query, the Cosmos DB service won't execute this operation and the API will throw a `DocumentClientException` exception with the `HttpStatusCode` property set to `429`. This HTTP status code means that the request made to Azure Cosmos DB has exceeded the provisioned throughput and it couldn't be executed.

In some cases, the only way to execute the request would be to increase the provisioned throughput. For example, if we have a single operation that requires more than 1,000 RU/s but we have provisioned only 1,000 RU/s, there will be no way to execute the operation unless we increase the provisioned throughput. No matter the number of times we retry, the operation will always fail. Of course, we should avoid operations that require a huge amount of RU/s.

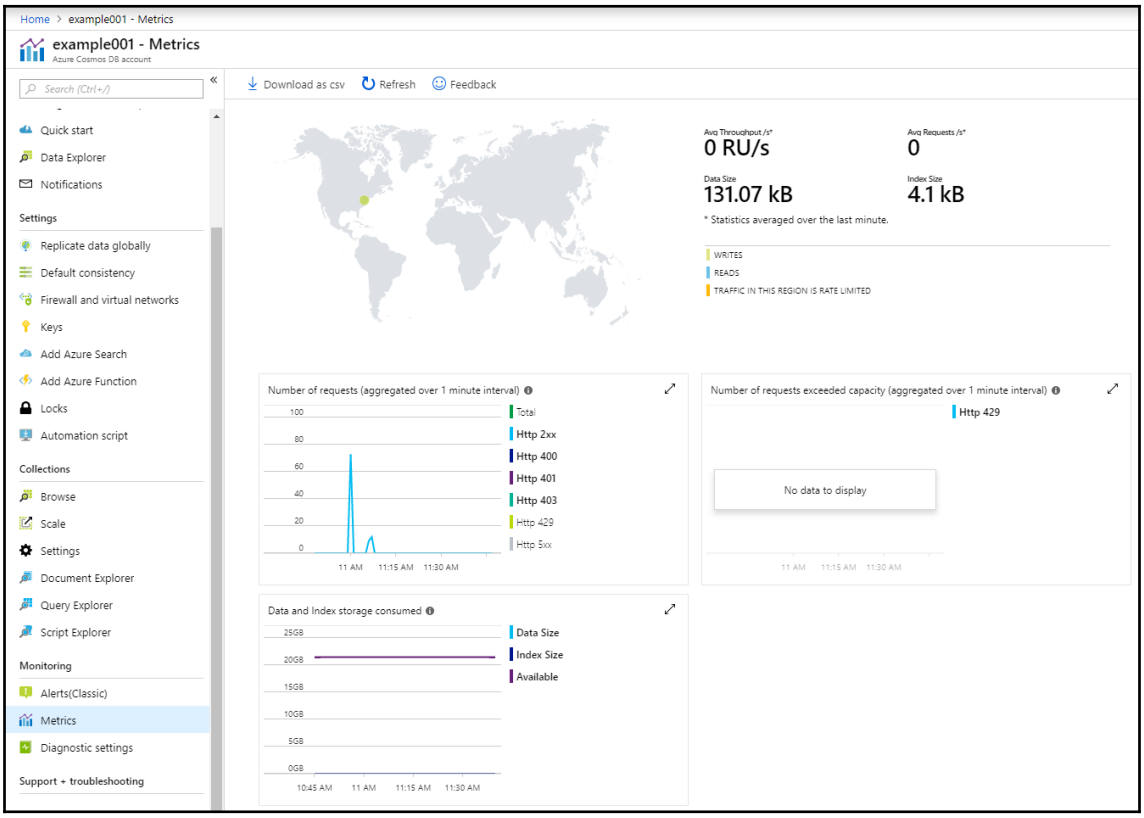
If we have two operations that require 501 RU/s each and we have provisioned only 1,000 RU/s, neither operation would be able to be executed in the same second as the other because they will consume 501 * 2 = 1,002 RU/s. One of them will fail and throw the previously explained exception. However, in this case, it would be possible to retry the operation that failed in the next second because the next second will have 1,000 RU/s available again and we only need 501 RU/s. Hence, if no other operation that requires more than 499 RU/s is executed concurrently, the operation that initially failed will succeed on the next retry.

There are many settings in the SDK that allow us to configure the number of retries that the SDK automatically performs for any operation that hits the provisioned RU/s rate limit before the `DocumentClientException` exception is thrown. Thus, we have to consider the behavior of the Cosmos DB service and the configuration of the SDK. There is no single configuration that fits all scenarios. It is necessary to decide which is the most convenient way of handling rate limiting and retries based on our specific application needs.

We can easily check the requests that exceeded provisioned throughput and generated an `HTTP 429` status code from the Azure Cosmos DB service with the Azure portal. Follow the next steps to check the metrics for the collection we are using in our sample application:

1. In the Azure portal, make sure you are in the page for the Cosmos DB account in the portal.
2. Click on the **Metrics** option.
3. Select the desired database from the **Database(s)** dropdown (`Competition`).
4. Select the desired collection from the **Collection(s)** dropdown (`Competitions1`).
5. Click **1 hour** at the right-hand side of the dropdowns to summarize the overview of the metrics for the last hour.
6. The **Overview** tab will display the following information about the selected database and collection for the last hour:
    - A map with the region
    - The average throughput in RU/s
    - The average number of requests per second
    - The data size
    - The index size
    - A chart with the number of requests and their status codes
    - A chart with the number of requests that exceeded capacity; that is, those requests that generated an `HTTP 429` status code
    - A chart with the data and index storage available and consumed

By inspecting the metrics, we can easily check whether we require more throughput or specific optimizations to reduce the RU/s consumed. The following screenshot shows an example of the metrics after running our sample application. Note that there are no requests that exceeded the provisioned throughput:



Tracking consumed request units with client-side code

There are many variables that determine the way Cosmos DB calculates the request unit charge for each operation. The first variable is the amount of data an operation or query reads or writes. 1 RU is how much effort it takes to read 1 KB of data from Cosmos DB that directly references the document with its URI or self link. Writes are more expensive than reads because they require more resources. The amount of properties and data you have in a document affects the cost as well. The data consistency levels, such as strong or bounded staleness, can cause more reads. Indexes affect your query costs. Your query patterns and the finally stored procedures and triggers you defined will add more request units with more complicated query executions. These are all factors that can be optimized, fine-tuned, and monitored.

Now we will establish a breakpoint in one of the methods that reads and updates a scheduled competition with platforms to inspect the properties provided by the `DocumentResponse` and `ResourceResponse` instances for the read and update operations. First, make sure you remove all the existing breakpoints.

Go to the following line within the `UpdateScheduledCompetitionWithPlatforms` static method:

```
readCompetitionResponse.Document.DateTime = newDateTime;
```

Right-click on the line and select **Breakpoint | Insert breakpoint** in the context menu.

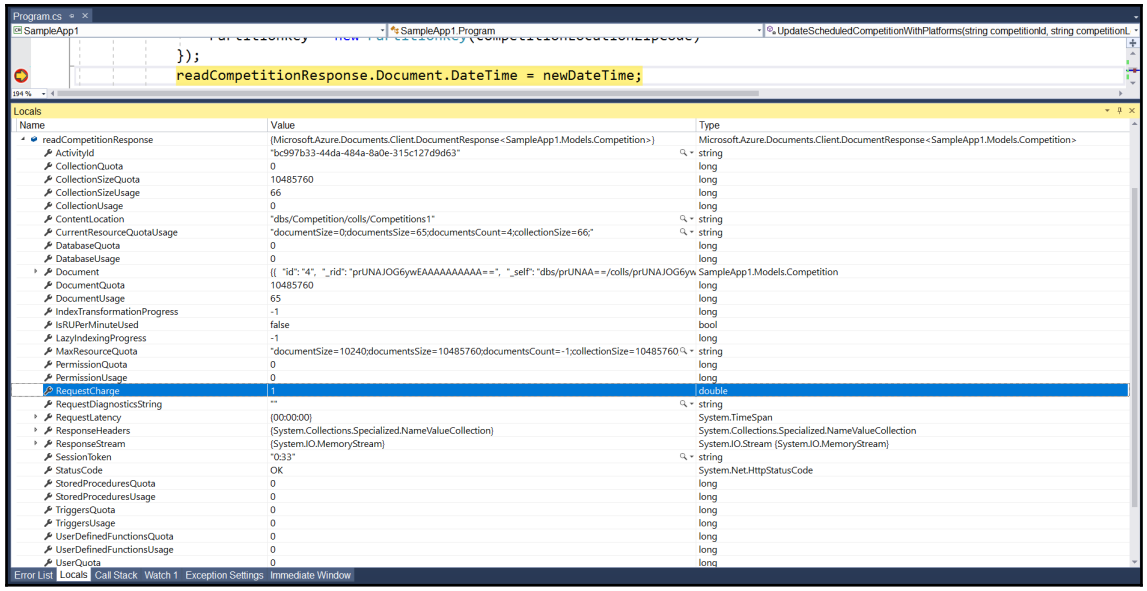
Now go to the following line within the same method:

```
if (updatedCompetitionResponse.StatusCode == System.Net.HttpStatusCode.OK)
```

Right-click on the line and select **Breakpoint | Insert breakpoint** in the context menu.

Start debugging the application.

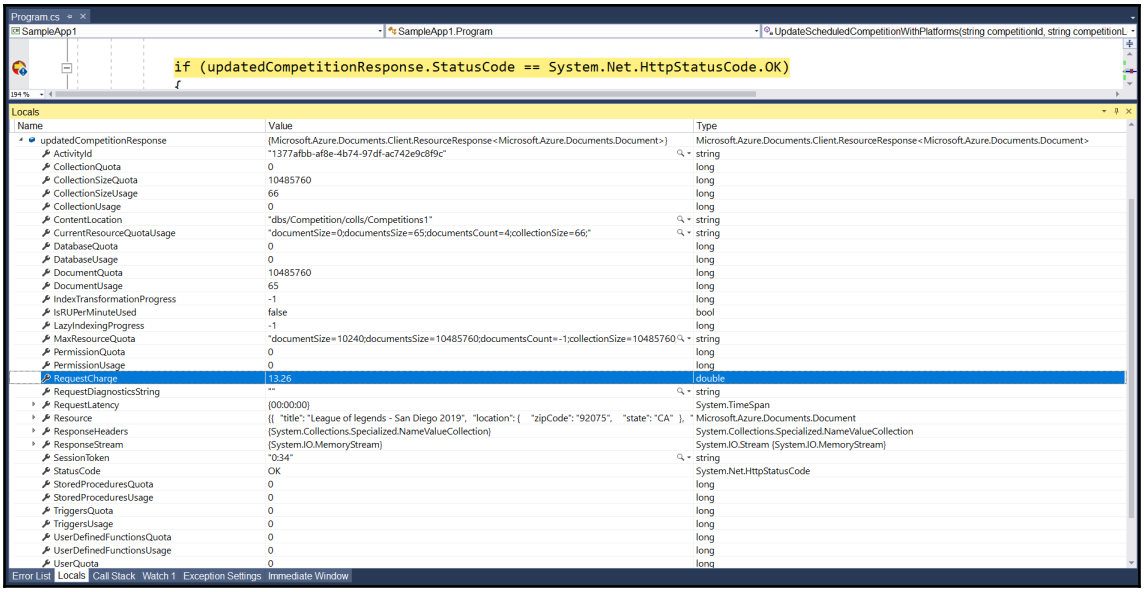
Inspect the value for the `readCompetitionResponse` variable in the **Locals** or **Auto** panel. This variable holds the `DocumentResponse<Competition>` instance with the result of the call to the `client.ReadDocumentAsync<Competition>` method. Expand `readCompetitionResponse` and check the different properties. The `RequestCharge` property provides us with the request units that have been charged for this document read operation: `1`. There are many other properties that also provide valuable information about the operation with the document, as well as properties that indicate values that apply to the document collection and the database. The following screenshot shows the properties, their values, and their types:



The `CollectionSizeQuota` property specifies the maximum size for the collection in which we have performed the `10485760` operation. The `CollectionSizeUsage` property indicates the size that we are consuming from the collection. These properties provide information that is related to the collection, and therefore, we can use them to check whether we are running out of space within a collection. There are many other properties that provide quotas and actual usage for different resources.

Continue debugging the application until it hits the next breakpoint.

Inspect the value for the `updatedCompetitionResponse` variable in the **Locals** or **Auto** panel. This variable holds the `ResourceResponse<Document>` instance with the result of the call to the `client.ReplaceDocumentAsync` method. Expand `updatedCompetitionResponse` and check the different properties. In this case, the `RequestCharge` property provides a value of `13.26`. However, remember that this value might be different in your configuration. A document replacement operation consumes more request units than a document read operation. The following screenshot shows the properties, their values, and their types:



So far, we have been working with the `ExecuteNextAsync` method to retrieve each of the pages for the results of the asynchronous queries. However, in order to check the request charge for asynchronous queries, we have to make a small edit to the existing code to be able to inspect the `FeedResponse` instance.

Now stop debugging the application and replace the code for the existing `GetCompetitionByTitleWithLinq` static method in the `Program` class with the following lines. Note that the added lines are highlighted in the next code snippet. The code file for the sample is included in the `learning_cosmos_db_06_01` folder in the `SampleApp3/SampleApp1/Program.cs` file:

```
private static async Task<Competition> GetCompetitionByTitleWithLinq(string
title)
{
    // Build a query to retrieve a Competition with a specific title
    var documentQuery =
client.CreateDocumentQuery<Competition>(collectionUri,
        new FeedOptions()
        {
            EnableCrossPartitionQuery = true,
            MaxItemCount = 1,
        })
        .Where(c => c.Title == title)
        .Select(c => c)
        .AsDocumentQuery();
    while (documentQuery.HasMoreResults)
    {
        var feedResponse = await
documentQuery.ExecuteNextAsync<Competition>();
        foreach (var competition in feedResponse)
        {
            Console.WriteLine(
                $"The Competition with the following title exists:
{title}");
            Console.WriteLine(competition);
            return competition;
        }
    }

    // No matching document found
    return null;
}
```

The edited code saves the results of calling the `documentQuery.ExcuteNextAsync<Competition>` method in the `feedResponse` variable. Then, the next line uses `feedResponse` to get its enumerator and retrieve each `Competition` instance.

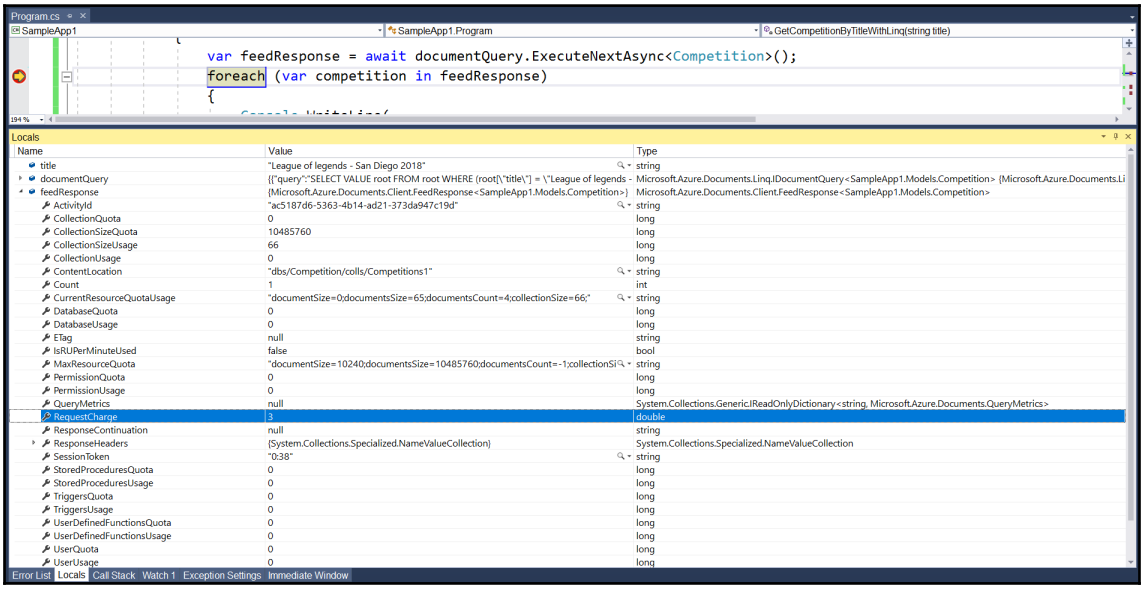
Go to the following line within the `GetCompetitionByTitleWithLinq` static method:

```
foreach (var competition in feedResponse)
```

Right-click on the line and select **Breakpoint | Insert breakpoint** in the context menu.

Start debugging the application and continue debugging it until it hits the recently added breakpoint.

Inspect the value for the `feedResponse` variable in the **Locals** or **Auto** panel. This variable holds the `FeedResponse<Competition>` instance with the result of the call to the `documentQuery.ExecuteNextAsync<Competition>` method. Expand `feedResponse` and check the different properties. The `RequestCharge` property provides us with the request units that have been charged for this query: `3`. However, remember that this value might be different in your configuration. A query consumes more request units than a document read operation. The following screenshot shows the properties, their values, and their types:



We can write the code necessary for summing the request units charged to the different operations and queries we perform and have a better understanding of our required provisioned throughput.

Understanding the options for provisioning request units

So far, we have been provisioning throughput for each collection. This gives us more granular control on how many request units we will need per container. This option is usually better if you have a smaller number of containers and require guaranteed throughput on each container backed by SLA. Keep in mind that all physical partitions of a Cosmos DB container will equally share the number of request units available for the container.

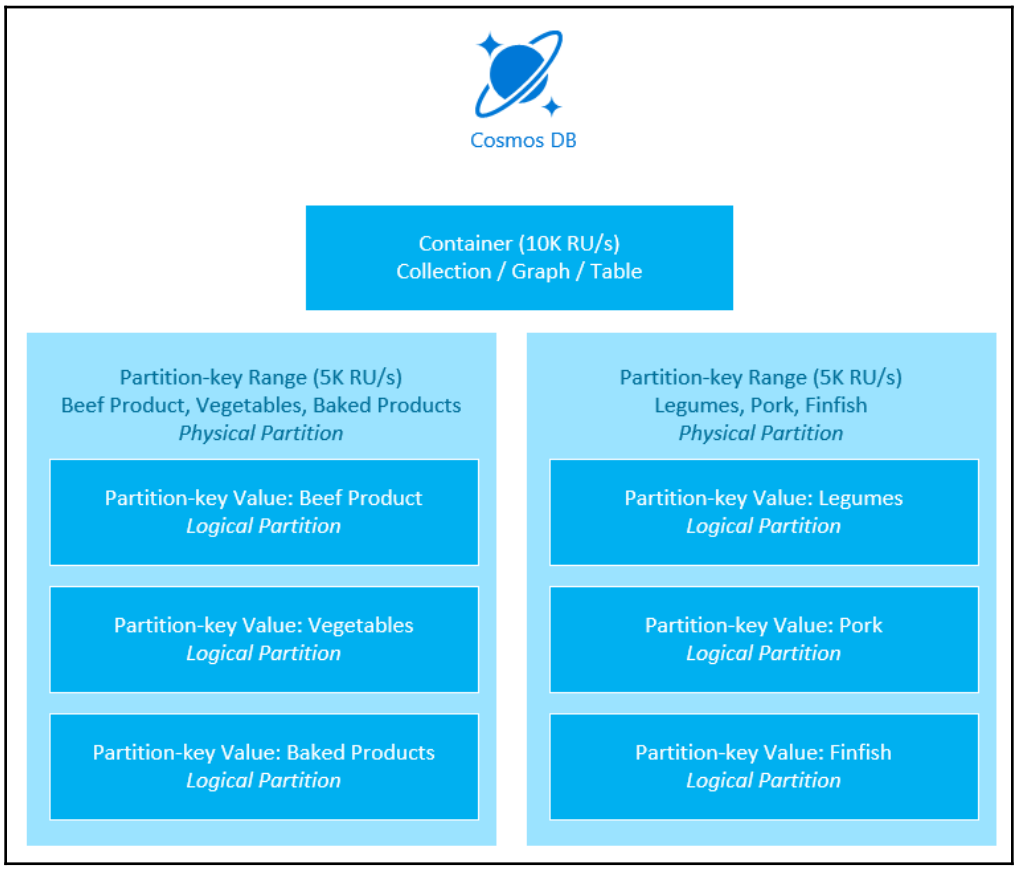
The other option is to provision throughput at the database level. In this case, all the containers within the database will share the total pool of request units you have. This can be a more comfortable management option when you have a high number of containers and do not necessarily want to manage all individually.

One crucial difference between container-level provisioning and database-level provisioning is the minimum throughput we can provision. For partitioned containers, the minimum provisioned throughput allowed is 1,000 RU/s. For database-level provisioning the minimum provisioned throughput is 50,000 RU/s.

Learning portioning strategies

To implement the right partitioning strategy for your needs, it is essential to understand how Cosmos DB does partitioning internally. In its usual sense, a partition may or may not exist. Physical and logical is how we have separated the two types of partitions. Out of all the partition key values we have in our dataset, Cosmos DB decides when we need a physical partition and moves an appropriate set of logical partitions to a separated physical partition when required. A physical partition consists of a reserved SSD storage area and variable compute resources. The management of physical partitions is done entirely by Cosmos DB. We do not have any control over it, except in picking the right partition key design to influence how request units might end up being distributed across physical partitions. The number of physical partitions on a container will vary based on the RU load on the container, and can increase up to the number of unique partition key values we have in a container.

In the following diagram, a container with two physical and six logical partitions shares RU/s allocations across physical partitions equally:

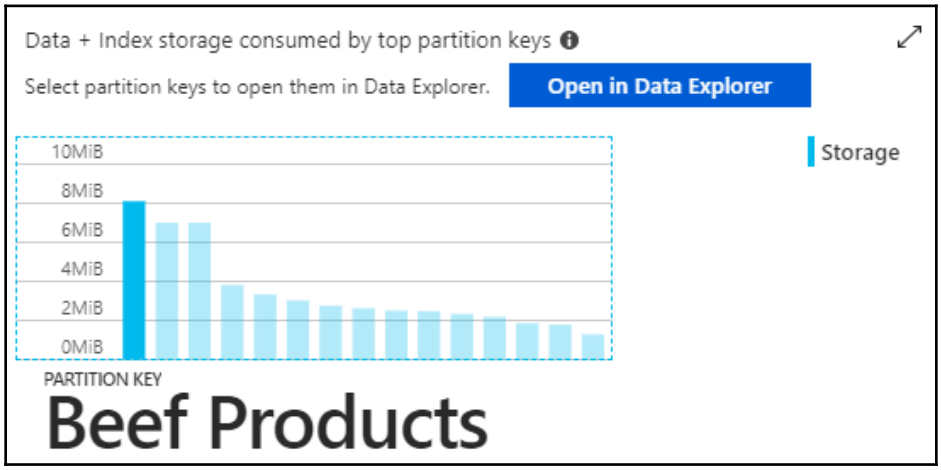


Logical partitions define every partition represented by a unique partition key in a dataset in a container. Multiple logical partitions can be in a single physical partition, depending on their RU load, or can be hosted in their reserved physical partitions.

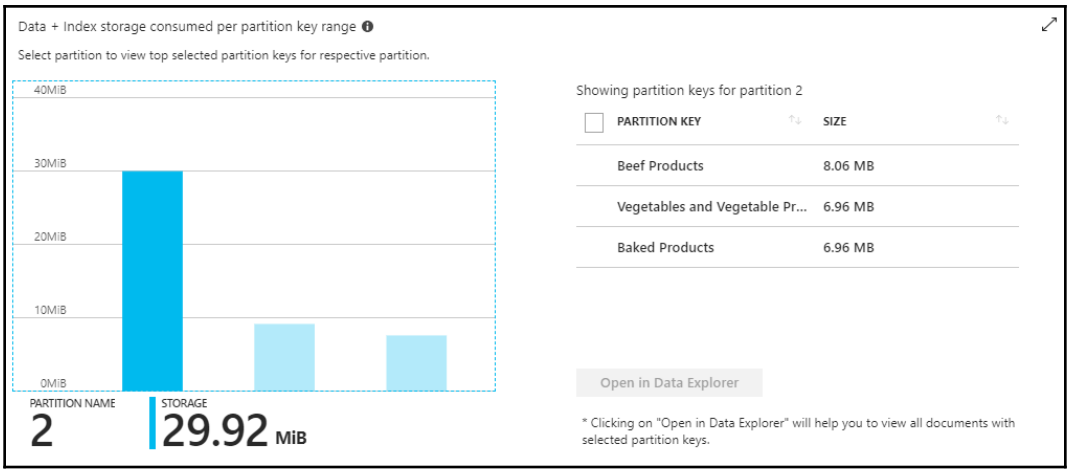
When a container is first provisioned for *X* number of RUs, Cosmos DB will look for consumed data and decide on how many physical partitions are needed. If RU consumption is higher than a single physical partition's maximum throughput, Cosmos DB will create as many partitions as needed to serve the *X* number of RUs required. Every physical partition will get an equal share of the total RUs provisioned for the container. Logical partitions will be distributed across the number of physical partitions.

The choice of the perfect partition key is critical for a scalable, long-lasting Cosmos DB implementation. When a single logical partition is stuck in a separate physical partition, there will be no way to scale it further. For example, at the time I was writing this book, a single physical partition in Cosmos DB had a limit of 10 GB of data storage. This limit could only be surpassed by adding new partition key values into the container so that Cosmos DB can create a new physical partition with the new logical partition. It is not just about the storage; compute needs to be evenly distributed as well. The amount of RUs provisioned for a container will be equally shared by all physical partitions. If you need 5,000 RUs in a partition called `Seattle` and have a total of 10 physical partitions, you will need to scale up to 10 * 5,000 RU to make sure your `Seattle` partition gets its fair share where your other partitions might only need 1,000 RUs.

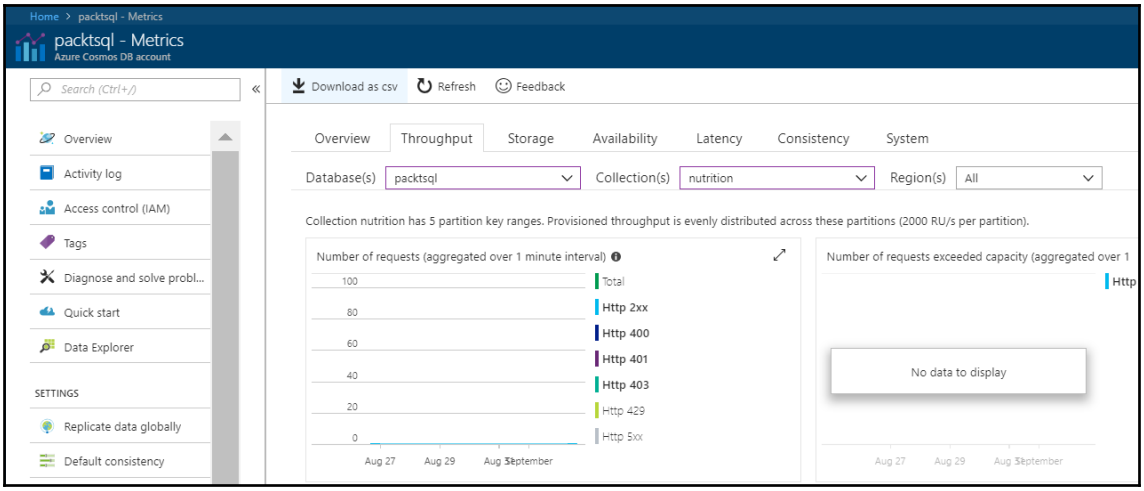
We already analyzed how to monitor RU consumption and data storage in the **Metrics** section of the Azure portal. The next screenshot shows how much storage space is used by top partitions, including the data size and the index size. Even though the numbers in this particular sample dataset are small, the fact that **Beef Products** has mostly three times the data as most other partitions is a red flag. If the incoming data keeps the same balance, we will hit the limit with this partition before other partitions fill up:



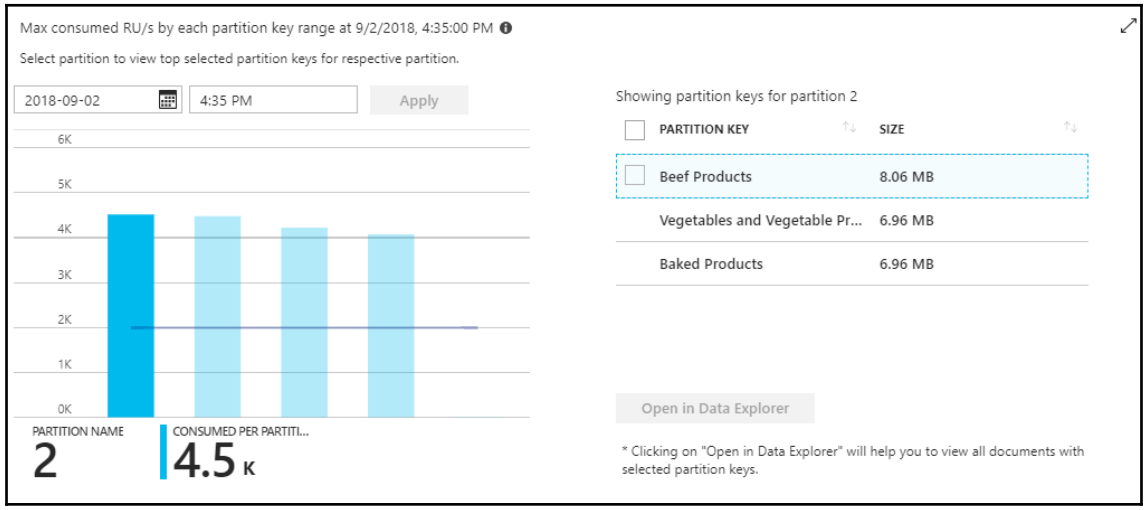
The next screenshot shows another chart from the Azure portal that displays the top partition key ranges. These are physical partitions hosting multiple logical partitions. In this example, you can see that the top physical partition, with 30 MB data in it, hosts logical partitions with the **Beef Products**, **Vegetables**, and **Baked Products** partition keys:



By navigating through the reporting screens in the **Throughput** page, you can find out how many physical partitions you have. In the next screenshot, you can read the **Provisioned throughput is evenly distributed across these partitions (2000 RU/s per partition)** statement. The `Nutrition` collection is currently set up for 10,000 RU/s, and because of the five partitions Cosmos DB had to create to serve enough RUs for all queries, the total RU is divided to provide each partition with 2,000 RU/s:



Looking at the **Throughput** metrics page, a critical graph is at the bottom of the screen titled **Max consumed RU/s by each partition key range**. In the next screenshot, we have four physical partitions, all exceeding their RU/s limits. The dark blue horizontal line in the graph represents the RU/s limit we have on every partition. It looks like all four partitions have exceeded their limits. The fact that all partitions had the same load is a good sign. We do not have a hot partition. What we have is an overall outage. If just one of the partitions had a high load, then it would be a hot partition issue where a single partition exceeds the limit. If that hot physical partition has multiple logical partitions, Cosmos DB would scale and create a new physical partition to distribute the load. If a single logical partition created a hot partition, there would be no way out other than scaling the full container to make sure the divided RUs per partition was higher than the hot partitions requirement. This is why a good, balanced design for partition key selection is crucial:



Partition key selection is not only for proper RU distribution and consumption. A lousy partitioning design can cause cross-partition queries, which are very costly and disabled by default.

Note that Cosmos DB throughput SLAs apply only to single-partition queries. Running cross-partition queries in Cosmos DB is not ideal and should be considered a last resort. We worked with cross-partition queries in our previous examples to demonstrate how we could run them.

When you run a cross-partition query, the SDK will fetch key ranges from Cosmos DB and run all single-partition queries in parallel to fulfill the requirement of the cross-partition query you wrote. Depending on the degree of parallelism, a cross-partition query may sometimes provide the same performance as a single-partition query, and sometimes it may not. Upper limits for parallel query execution in Cosmos DB can be configured with the `MaxDegreeOfParallelism` and `MaxBufferedItemCount` parameters within the SDK. `MaxDegreeOfParallelism` helps to increase the concurrent connection for a cross-partition query. By default, it is set to `0`.

What if you can't find a single property in your documents that can help you achieve a proper distribution of data and query load? You can combine multiple fields and create synthetic keys.

For example, imagine that you have the following document:

```
{
    "city": "seattle",
    "date": 2018
}
```

If you are not sure that using the `city` key as a partition key will be good enough, you can combine two fields and create a new one by concatenating the two values to be used as a partition key. For example, the following document combines `city` and `year` to create a `partitionKey` key that we will use as the partition key:

```
{
    "city": "seattle",
    "year": 2018,
    "partitionKey": "seattle-2018"
}
```

What if we just assign a unique value to every document and use that as the partition key? That sounds pretty flexible. The issue is that if you wanted to use cross-document transactions with stored procedures or triggers, the requirement would be that both documents be in the same partition.

By using unique values and having random GUIDs for all your documents as the partition key value, you will make it almost impossible to have multiple documents in the same partition. The container will be able to scale pretty well, but you will never know what documents can be used in a single transaction.

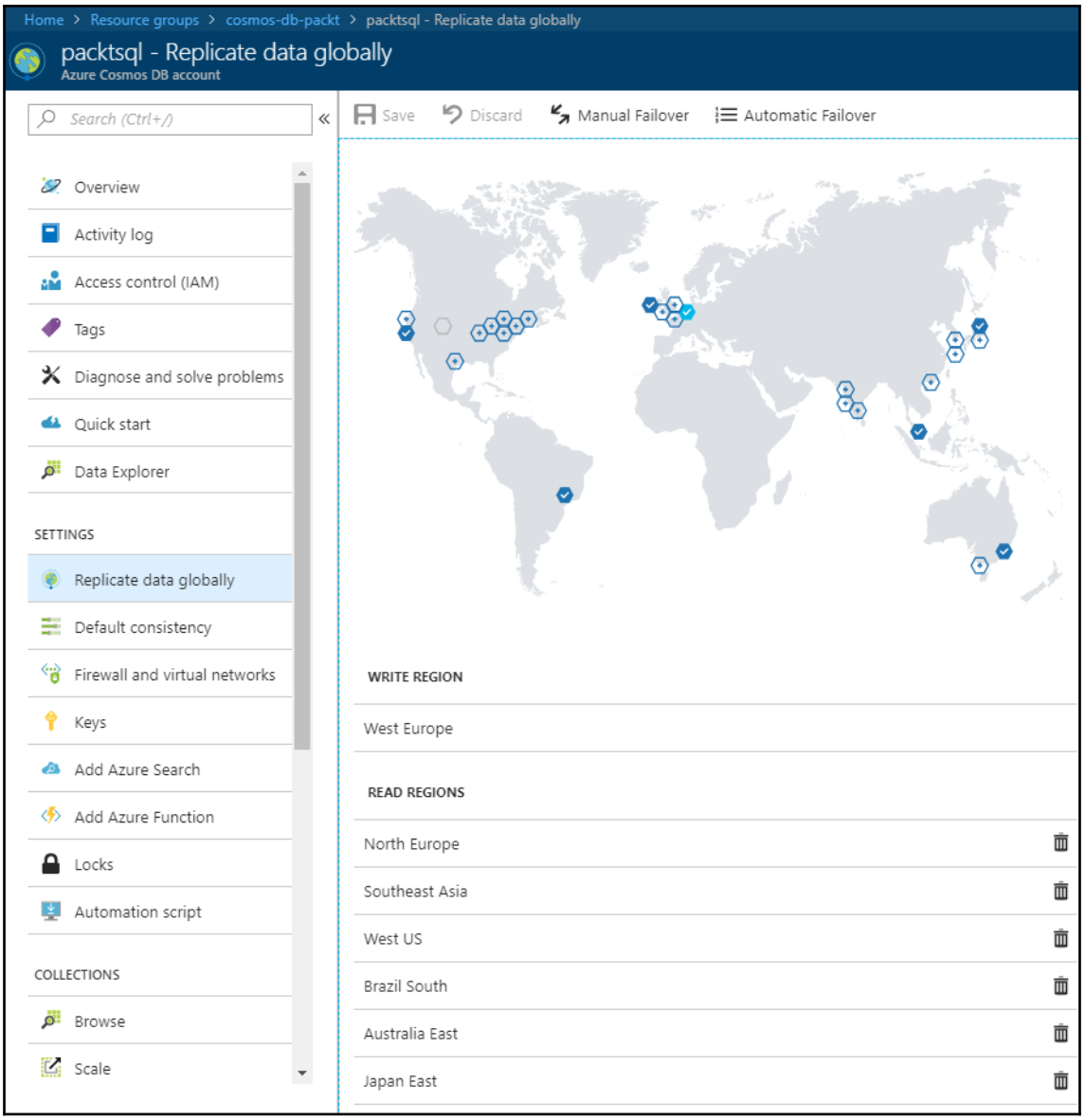
The obvious choices for a partition key are things like user ID, company ID, tenant ID, and device ID. Whatever ID you pick, you need to make sure you will never need more than the maximum performance and storage of a single physical partition; that will mean you will never hit any limits. Once that is accomplished, the second evaluation point would be the RU load distribution. A good practice would be to look into your application and figure out what common parameters exist in most used queries. Having those parameters in the partition key will make sure that you run single partition queries and distribute your load. You should consider your dataset saturation as well. If 75% of your customers are in the USA, picking the country field as a partition key is a recipe for a hot partition.

Deploying to multiple regions

The ability to operate at the global scale is one of the significant capabilities of Cosmos DB. It is not merely a database built for an on-premises server farm that has been moved into the cloud. Cosmos DB is a database built for the cloud from scratch. A cloud-native database needs to be global and be able to accommodate the needs of a global-scale application.

Deploying a multi-region database with Cosmos DB requires just a couple of mouse clicks or taps on the Azure portal. Navigate to the **Replicate data globally** blade of your Cosmos DB account in the Azure portal and start picking as many regions as you need. Once your selection is complete, hit **Save** and watch the magic happen. However, don't forget that deploying to multiple regions will have an impact on billing.

In the following screenshot, you can see that the global replication for Cosmos DB means selecting regions in the Azure portal. This will help to spread and scale your reads across multiple regions and help your applications read data from the nearest data center with the help of multi-homing:



The Cosmos DB SDK helps client applications decide their preferred read regions. When a request to a particular region fails, the client application will try the same request in the whatever region that follows in the preferred read regions list.

Understanding the five consistency levels

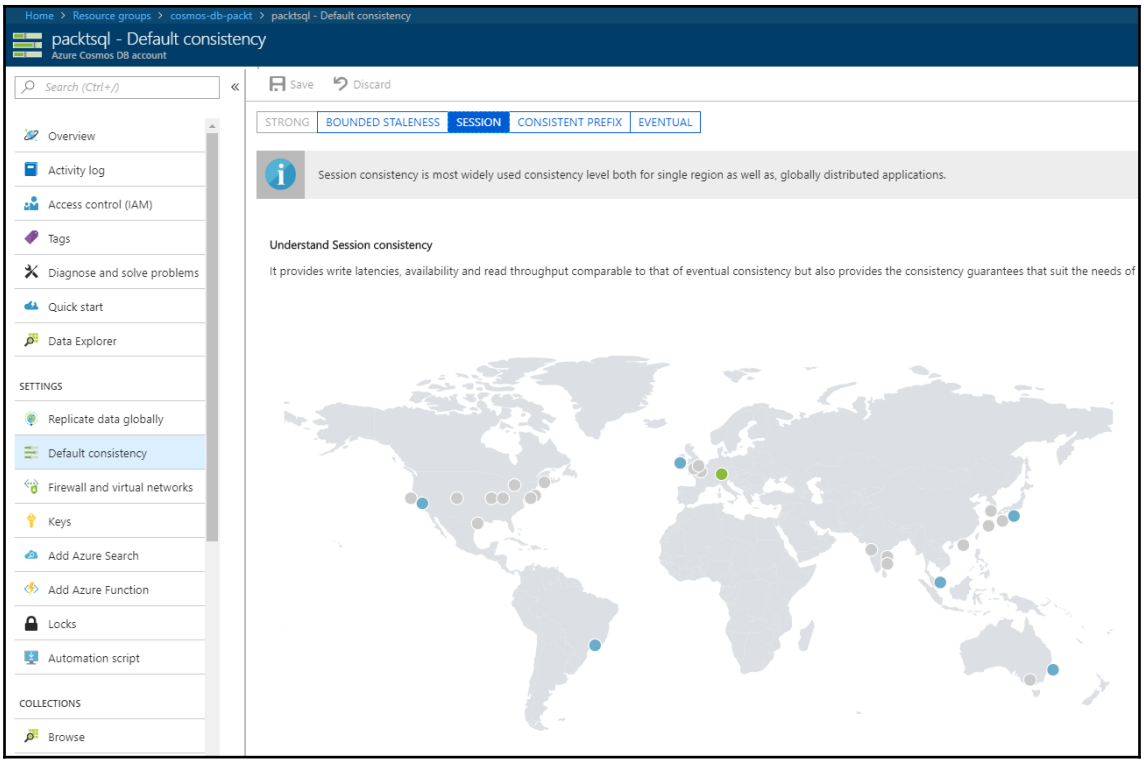
Now that we have multiple replicas of our data around the globe, what about data consistency? Cosmos DB provides the following five options:

- Strong
- Bounded staleness
- Session
- Consistent prefix
- Eventual

In a replicated dataset, there is always a trade-off between consistency, availability, throughput, and latency. In theoretical computer science, Brewer's theorem (named after computer scientist Eric Brewer), also known as CAP Theorem, says that *"a distributed database can only give two of the three guarantees; consistency, availability, partition tolerance"*. With this context in mind, Cosmos DB offers five different consistency levels, letting us fine-tune our priorities.

Keep in mind that high consistency levels will require more RUs as well.

The following screenshot shows the **Default Consistency** option in the Azure portal, which allows us to pick the desired consistency level:



For those of you experienced with distributed databases, strong and eventual consistency might sound familiar. Strong consistency is where everything is in perfect synchronization. All read operations get the latest data. A write operation never becomes visible until all replicas synchronously commit the data change. If you pick strong consistency, you can only add one region to your Cosmos DB account. In that case, what replicas are we talking about? Let's clarify that. Every physical partition set in Cosmos DB is replicated four times across four resource partitions. These are called replica sets. These copies are local to the region and help achieve multiple SLAs. On the other hand, the replications between regions are asynchronous. It would be very costly to provide strong consistency across regions. Considering the SLAs Cosmos DB is trying to hit, it makes sense that strong consistency is only available in single-region accounts.

What about the global database? How is that possible with single-region accounts? Of course, it is not. We will need to loosen up the consistency level a little bit – good thing that we've got four other consistency levels in our list.

Bounded staleness is somewhere between session consistency and strong consistency. It allows reads to lag behind writes for a specific amount of time or versions. This consistency level provides total global order. This is like strong consistency but with some lag. At this level, you can have as many Azure regions in your Cosmos DB account as you want. Regarding RU consumption, bounded staleness will take its toll on your operations. The cost of queries will be higher than session consistency, but the same as strong consistency.

Session consistency provides a perfect read-your-own-writes environment. It offers strong consistency scoped into a client session. The cost for session consistency is lower than bounded staleness but higher than eventual consistency.

Consistent prefix is between session consistency and eventual consistency. Consistent prefix makes sure read operations always return a prefix of all data updates without any gaps.

Eventual consistency is the weakest consistency level in Cosmos DB. In this level, when a client sends requests across multiple regions, it might get old data compared to what it received in the past from another region. Eventual consistency provides the lowest latency and RU consumption.

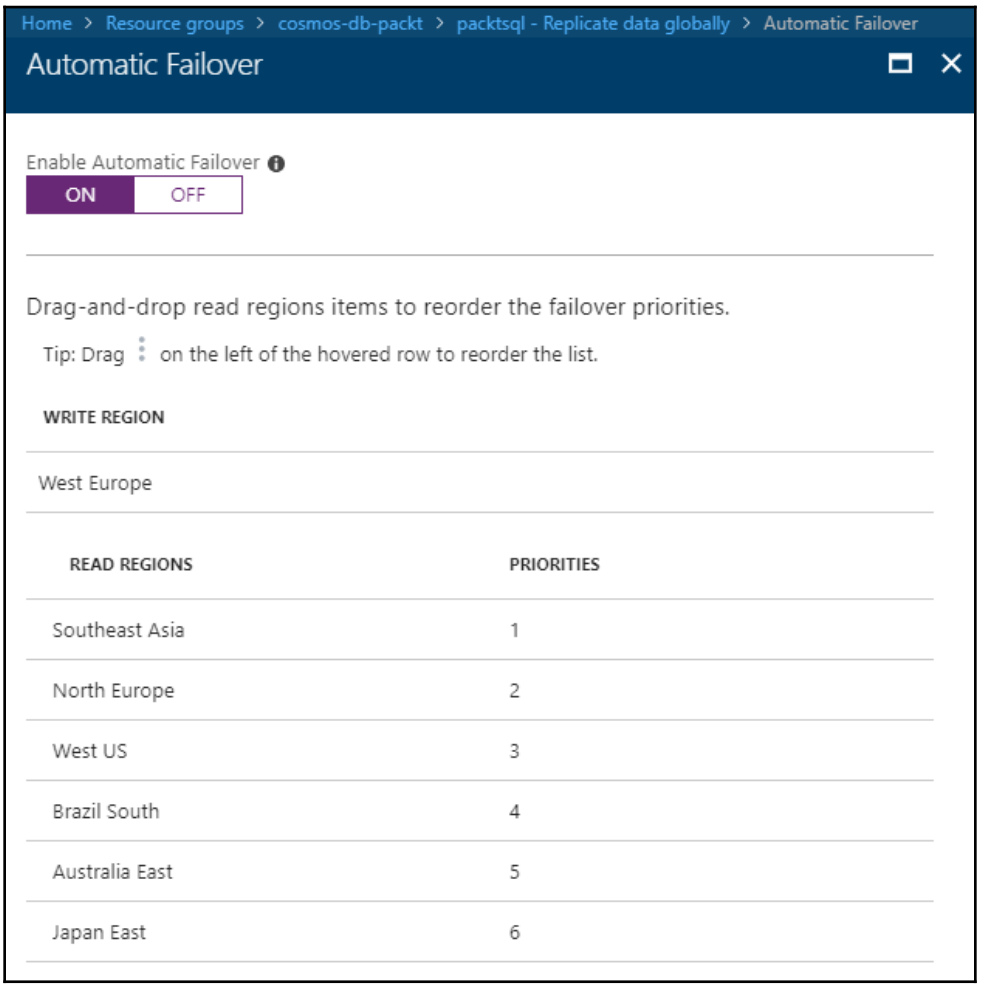
Consistency levels for a container can be set in the Azure portal in the **Default consistency** blade. You can override consistency levels per query depending on the operation. For example, you can ask for a lazy read with a low RU cost even if the default consistency level is higher.

Taking advantage of regional failover

You might have noticed that on the **Replicate data globally** page in the Azure portal, there is only one region defined as **Write region**. This is because Cosmos DB is designed to get all writes into a single region and distribute the reads.

Cosmos DB started supporting multiple write regions with a feature called Multi-Master in a preview version during the last quarter of 2018. With a single write region, it is important to plan for regional failovers. When a read region fails, another region will take over. When a write region fails, you might want to prioritize what region to take over explicitly. To do so, you can find an **Automatic Failover** button on top of the region replication blade. Once you open up the page, you can enable automatic failover and drag and drop regions to create your prioritized failover list.

The following screenshot shows a sample configuration for the read regions and its priorities:



In addition to automatic failovers, you can change the write region to another region by simply triggering a manual failover. The manual failover button is available in the region management blade. Once you click on it, you will have a list of regions to failover the write region to. During a failover, there is no code change required in client applications. For some global applications, it makes sense to failover the write region to different regions at different times of the day to make sure writes go to the nearest location. Manual failover can be automated through Cosmos DB SDKs if needed.

Understanding indexing in Cosmos DB

The default configuration for indexing in Cosmos DB makes indexing happen automatically. Hence, whenever we create or update a document in a document collection, all the keys included in the document are indexed. This might sound counter-intuitive, but it is how the system is designed to work. No need for index management, unless you want to optimize your costs better or you require specific queries.

Keep in mind that every index you have in your dataset will have its toll on request units consumed and storage space used. Hence, if you are indexing keys that you are never going to use in search criteria, you are wasting resource units in every write operation.

In contrast, sometimes removing an index can increase the request unit cost of a query as well. Thus, it is very convenient to make sure that we don't remove indexes for keys that are included in search criteria. It is vital to use indexing strategically to come up with the best implementation. Let's look at what options Cosmos DB has to offer.

Cosmos DB has the following three index update modes:

- **Consistent index**: This is the default mode. As the name suggests, this indexing mode helps to keep an always consistent index. This mode of indexing will have its share of load, especially during writes. In this mode, the index is updated synchronously as part of the operation that persists or deletes a document in a collection. However, as soon as the operation has finished, the document is indexed and it can be queried immediately.
- **Lazy indexing**: This indexing mode updates the index asynchronously when the collection provisioned throughput is not fully utilized. The big risk of this indexing mode is that documents might be indexed slowly when the provisioned throughput is being consumed at high rates by all the operations. Queries might provide results that aren't consistent. For example, a `COUNT` query with specific criteria won't include the documents that aren't indexed.
- **None**: There is no indexing at all. This mode is only useful when we work with documents that are accessed by ID and we don't need to execute queries. Hence, we should only consider this option when we use a collection as key-value storage. If we run any query in a collection that isn't indexed, it is necessary to set the `EnableScanInQuery` property to `true` in the `FeedOptions` instance passed as an argument to the `CreateDocumentQuery` method. However, these queries will be executed as full scans that will consume an important amount of resource units.

Cosmos DB has the following three different indexing types that are suitable for diverse data types:

- **Hash**: This index type is mainly used for equality and `JOIN` queries. The data type for hash indexes can be `String` or `Number`.
- **Range**: This index type comes with the maximum index precision by default. This index type is used for range queries, equality queries, and sort operations (`ORDER BY`). Range indexes support `String` or `Number` as well. In Cosmos DB, `DateTime` values are stored as ISO 8601 strings, and therefore, range indexes help with range queries related to `DateTime` keys as well.
- **Geospatial**: In this index type, `Point`, `Polygon`, or `LineString` indexes are compatible with GeoJSON. Geospatial indexes support spatial queries and many spatial operations on the indexed types.

If we customize the indexing policies but we don't pick the right index types, our queries might get limited. For example, we can't use range operators in a query if the field does not have a range index. We can always force to run a query with a full scan by setting the explained `EnableScanInQuery` property to `true` in the `FeedOptions` instance passed as an argument to the `CreateDocumentQuery` method. However, we will always want to avoid full scan queries to reduce the RU charge.

Range and hash indexes can be further fine-tuned with an index precision parameter. This parameter helps us balance the storage overhead for the index and query performance. For numbers, the default precision is `-1` (maximum). Yeah, it sounds crazy, but `-1` is the maximum precision. If you increase the value, the index data size will decrease, but queries will need to scan more documents because the index record will point to a broader range of documents.

For string ranges, the precision has more effect because of the size of data for a single key. However, in order to be able to do sort queries (`ORDER BY`) for string keys, the precision needs to be `-1` (maximum).

Checking indexing policies for a collection with the Azure portal

An index policy specifies the indexing mode for a collection and includes a list of paths to index, or to exclude. The `includedPaths` key lists all the indexes in a container, the index types to be used, matching data types, and the index precision. Indexing policies can be manipulated on the fly by editing them on the Azure portal or with the Cosmos DB SDK.

Now we will use the Azure portal to check the indexing policy for the `Competitions1` document collection.

In the Azure portal, make sure you are in the page for the Cosmos DB account in the portal. Click on the **Data Explorer** option, click on the database name you used in the configuration for the examples in previous chapters (`Competition`) to expand the collections for the database, and click on the collection name you used for the examples (`Competitions1`). Click on **Scale & Settings** and scroll down to the JSON document shown under **Indexing Policy**. The following lines show the JSON document that defines the indexing policy for the collection:

```
{
    "indexingMode": "consistent",
    "automatic": true,
    "includedPaths": [
        {
            "path": "/*",
            "indexes": [
                {
                    "kind": "Range",
                    "dataType": "Number",
                    "precision": -1
                },
                {
                    "kind": "Hash",
                    "dataType": "String",
                    "precision": 3
                }
            ]
        }
    ],
    "excludedPaths": []
}
```

Test your knowledge

Let's see whether you can answer the following questions correctly:

1. On what basis does Azure bill provisioned RUs?
    1. Per-day basis
    2. Per-hour basis
    3. Per-second basis
2. How many RUs does it cost to read 1 KB of data from Cosmos DB directly referencing the document with its URI or self link?
    1. 1 RU
    2. 10 RUs
    3. 1,000 RUs
3. Which of the following numbers define the maximum precision for a Cosmos DB index?
    1. -1
    2. 256
    3. 65535
4. If a collection isn't indexed but you still want to run a query, which of the following properties must be set to `true` in the `FeedOptions` instance, which specifies the feed options for the query?
    1. `EnableFullScan`
    2. `EnableNonIndexedCollectionQuery`
    3. `EnableScanInQuery`
5. If a collection has 10,000 RUs provisioned and you have five physical partitions, how many RUs can be consumed on a single partition?
    1. 50,000 RUs
    2. 10,000 RUs
    3. 2,000 RUs

Summary

In this chapter, we learned to analyze many aspects of Cosmos DB that allow us to design and maintain scalable architectures. We used our sample application to understand how many important things work, and we worked with the other examples to understand complex topics related to scalability.

Answers

Chapter 1: Introduction to NoSQL in Cosmos DB

1. **1**
2. **2**
3. **3**
4. **3**
5. **2**

Chapter 2: Getting Started with Cosmos DB Development and NoSQL Document Databases

1. **1**
2. **3**
3. **3**
4. **2**
5. **3**

Chapter 3: Writing and Running Queries on NoSQL Document Databases

Right answers

1. **3**
2. **1**
3. **1**
4. **3**
5. **2**

Chapter 4: Building an Application with C#, Cosmos DB, a NoSQL Document Database, and the SQL API

1. **2**
2. **1**
3. **2**
4. **3**
5. **1**

Chapter 5: Working with POCOs, LINQ, and a NoSQL Document Database

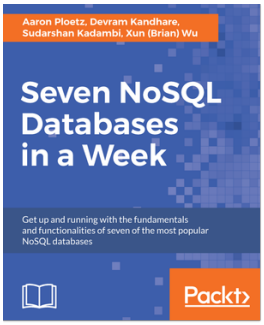
1. **3**
2. **1**
3. **1**
4. **2**
5. **2**

Chapter 6: Tuning and Managing Scalability with Cosmos DB

1. **2**
2. **1**
3. **1**
4. **3**
5. **3**

Other Books You May Enjoy

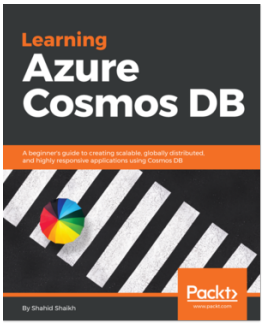
If you enjoyed this book, you may be interested in these other books by Packt:



Seven NoSQL Databases in a Week
Aaron Ploetz

ISBN: 9781787288867

- Understand how MongoDB provides high-performance, high-availability, and automatic scaling
- Interact with your Neo4j instances via database queries, Python scripts, and Java application code
- Get familiar with common querying and programming methods to interact with Redis
- Study the different types of problems Cassandra can solve
- Work with HBase components to support common operations such as creating tables and reading/writing data
- Discover data models and work with CRUD operations using DynamoDB
- Discover what makes InfluxDB a great choice for working with time-series data



Learning Azure Cosmos DB
Shahid Shaikh

ISBN: 9781788476171

- Build highly responsive and mission-critical applications
- Understand how distributed databases are important for global scale and low latency
- Understand how to write globally distributed applications the right way
- Implement comprehensive SLAs for throughput, latency, consistency, and availability
- Implement multiple data models and popular APIs for accessing and querying data
- Implement best practices covering data security in order to detect, prevent and respond to database breaches

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index