Processing highly connected Data using Azure Cosmos DB and Gremlin

Introduction

In this article, we take a look at processing graph-oriented data using Azure Cosmos DB. A graph allows us to represent connections between entities in a convenient and natural way. In many data processing scenarios, we want to efficiently store and analyze huge amounts of these connections.

For example, think about social networks like Twitter or Facebook: They basically consist of a vast number of people that are connected to each other in various ways. By analyzing the connections between them, we can infer social relationships, optimize content delivery and make meaningful content suggestions to users.

There are a lot of other problems that can be solved using graphs: Route optimization in logistics, automated product recommendation or monitoring of network topologies, to name a few.

Azure Cosmos DB enables us to efficiently store and analyze highly connected data using graph structures. Graphs can be read and manipulated using **Gremlin**, a popular graph traversal language originated from the Apache TinkerPop project. As Cosmos DB supports seamless scaling and multi-region replication of graphs, it is well-suited for large-scale data processing scenarios.

In this article, we will do the following:

- We will learn more about graph databases and Cosmos DB
- We utilize Gremlin queries to construct a graph and store it inside Cosmos DB

- We take a look at some of Gremlin’s query capabilities
- Finally, I want to discuss the usage of graph databases alongside other database systems. This is a very important aspect when designing larger systems, as you may want to take advantage of graph database capabilities but still use other databases, e.g. a relational database.

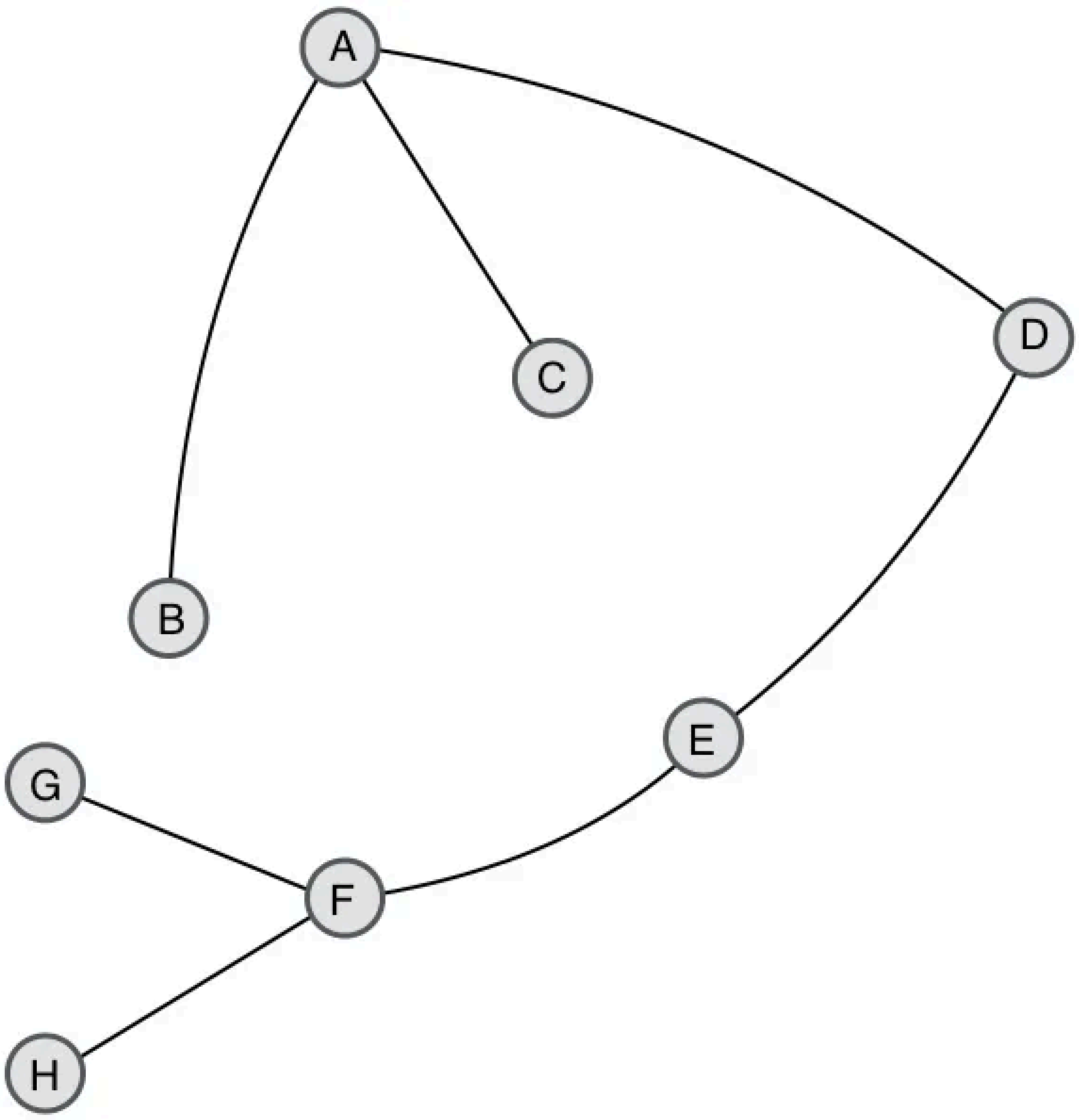
Let’s jump right in. ✨

About Vertices, Edges and Graph Databases

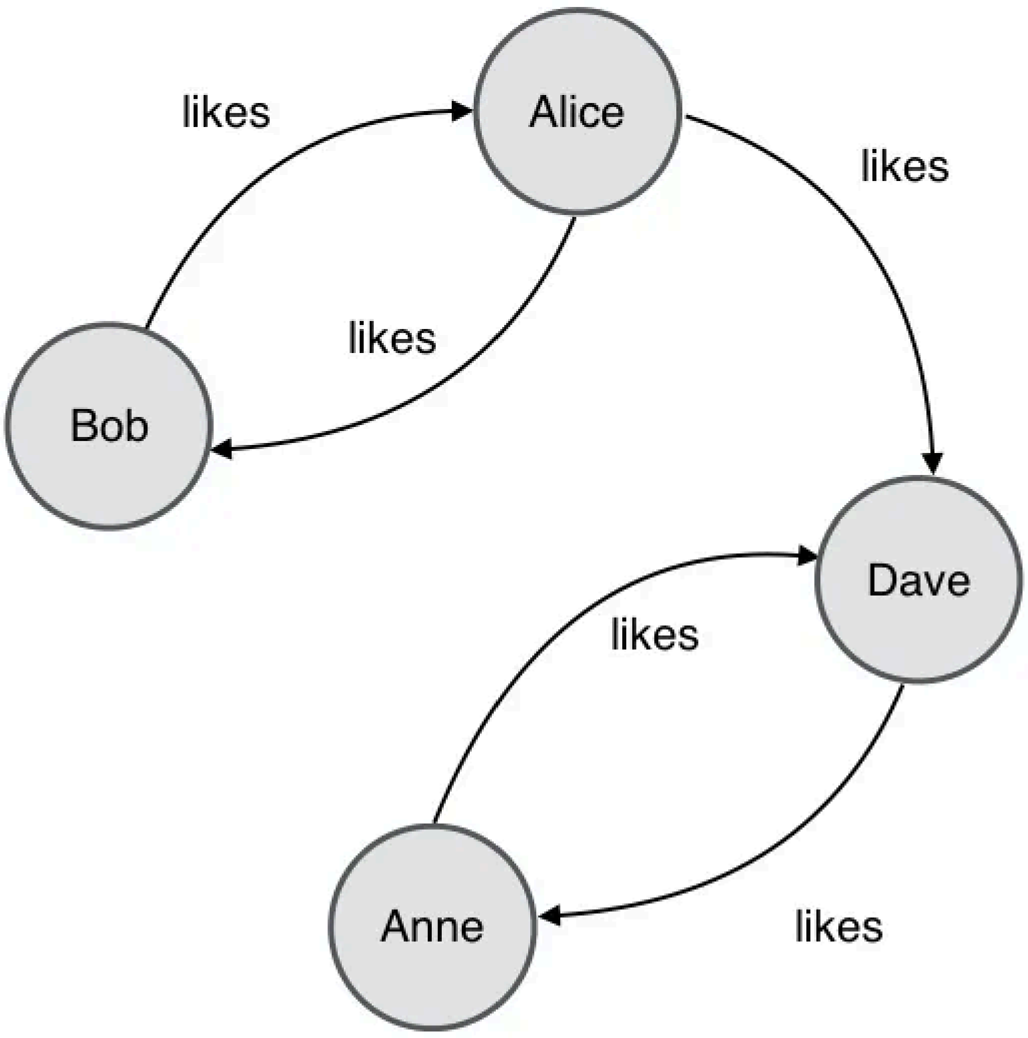
A graph consists of objects and a set of relationships between them. From an abstract, mathematical point of view, the objects are called **vertices** and the relationships are called **edges**.

An edge connects a pair of vertices and defines a relationship between them. In an **undirected graph**, the edges do not have a direction. Thus, when we have a pair of vertices named x and y, the edge (x, y) is identical to the edge (y, x).

In a **directed graph**, edges do have a direction. Below are two graph visualizations as an example:



A simple, undirected graph



A simple, directed graph visualizing a sociogram

By traversing (or following) the edges of these graphs, we are able to answer the following questions:

- Is there any path from A to F in the undirected graph?
  What is the shortest path from C to E?
- Which person is most popular (is liked most) in the above sociogram?

A **graph database** is a database that represents and stores data using graphs, that is to say vertices and edges. The graph can be enriched with arbitrary semantic information, stored as a set of properties. **Properties** are key/value-elements that are either attached to a vertex or an edge. Just like in a document-oriented database, the properties do not follow a predetermined schema.

Queries against a graph database are expressed in a language which is suitable for traversing graphs. The efficient execution of graph traversals is one big strength of graph databases. By contrast, in a relational database, this is a task that does not scale very well.

If we were to implement a sociogram describing friendships using an RDBMS, we might come up with a table containing people and a table containing friend-relationships:

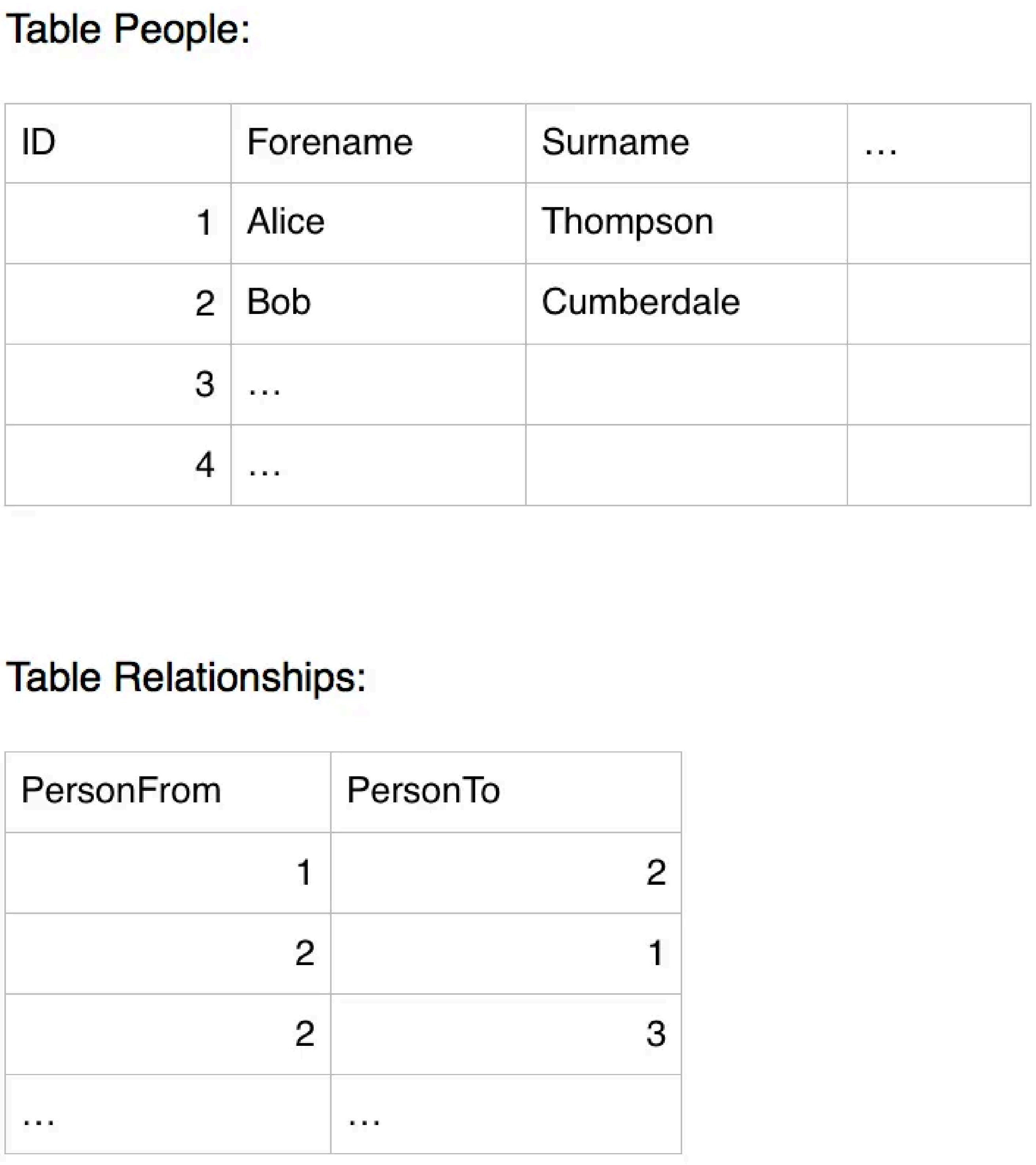


Table People:

ID	Forename	Surname	…
1	Alice	Thompson	
2	Bob	Cumberdale	
3	…		
4	…		

Table Relationships:

PersonFrom	PersonTo
1	2
2	1
2	3
…	…

The representation of a sociogram within an RDBMS

Traversing the graph now basically results in recursive JOINs: Do any friends of Alice, living in Springfield, cultivate a friendship with people living in Shelbyville? To answer this question, we have to fetch all relationship records involving Alice, fetch the records of her friends living in Springfield, fetch relationship records involving them, … and so on.

The query execution time in such scenarios easily gets unpredictable, let alone the complexity of expressing them. Of course we could invest some time to optimize these queries a bit by altering above table structure. However, we had to invest that work due to the fact that relational databases do not offer us any good tools to handle highly connected data. The above scenario is a typical example of **accidental complexity**, a term coined by Fred Brooks in his famous “No Silverbullet” paper. It is the technical complexity that is not relevant to the problem, often resulting from the wrong choice of language and tools.

By contrast, using Gremlin, fetching Shelbyville-based friends of Springfield-based friends of Alice might look like this:

```
g.V()
    .has('name', 'Alice').out()
    .has('city', 'Springfield').out()
    .has('city', 'Shelbyville')
```

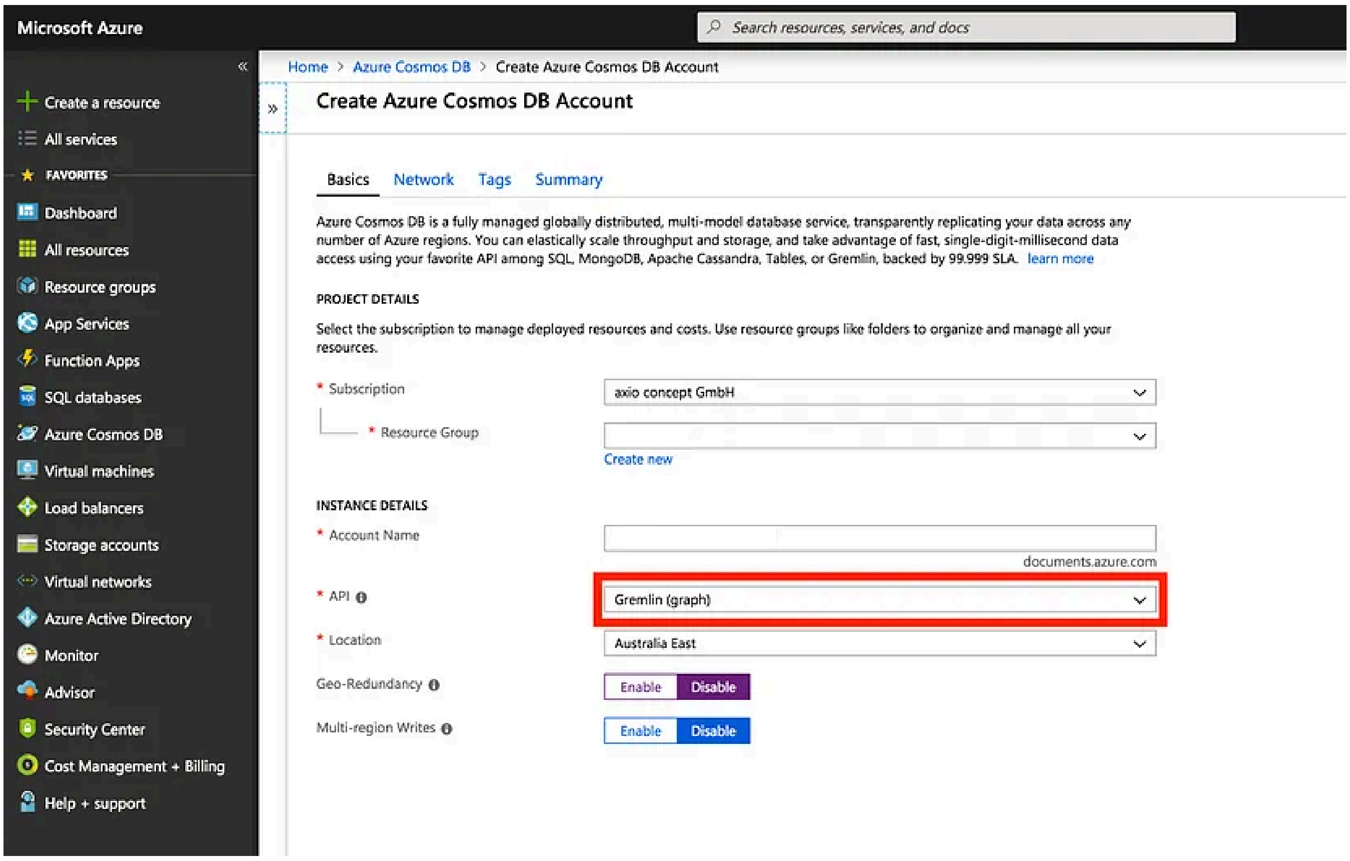
First Steps with Cosmos DB and Gremlin

Microsoft Azure Cosmos DB is a globally distributed, multi-model database platform that supports storage of documents, key-value collections and graphs. It offers elastic scaling for storage and throughput as well as multi-region replication.

In this section, we will create a Cosmos DB graph database and use the Data Explorer from the Azure Portal to run some basic Gremlin commands.

Creating a Graph Database

First, start by creating a new Cosmos DB account within the Azure Portal. Do not forget to select the Gremlin API as the instance API:



Create Azure Cosmos DB Account

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using your favorite API among SQL, MongoDB, Apache Cassandra, Tables, or Gremlin, backed by 99.999 SLA. learn more

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription: axio concept GmbH
* Resource Group:
Create new

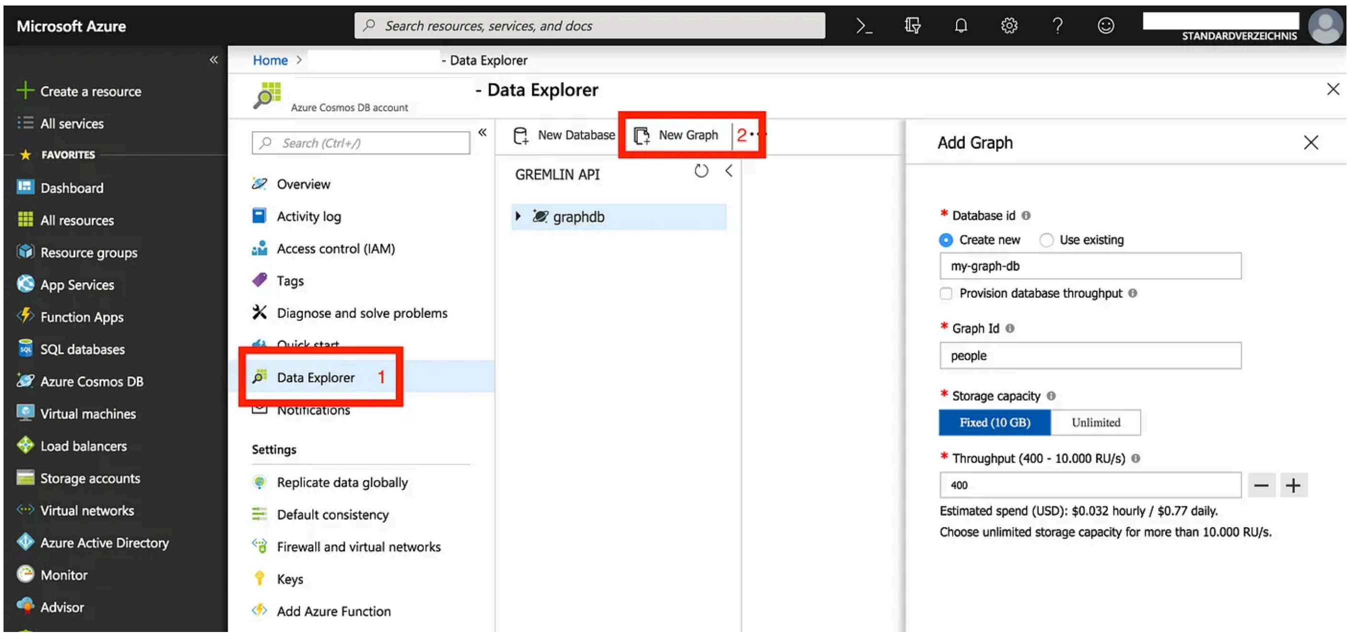
INSTANCE DETAILS

* Account Name:
documents.azure.com
* API: Gremlin (graph)
* Location: Australia East
Geo-Redundancy: Enable [✓ Disable]
Multi-region Writes: Enable [✓ Disable]

Creating a new Cosmos DB Account

Switch to your newly created Cosmos DB account and navigate to the **Data Explorer**. The data explorer gives you an overview of existing databases and graphs. It also can be used to visualize graphs and even manipulate vertices, edges and their properties.

Add a new graph as shown below:

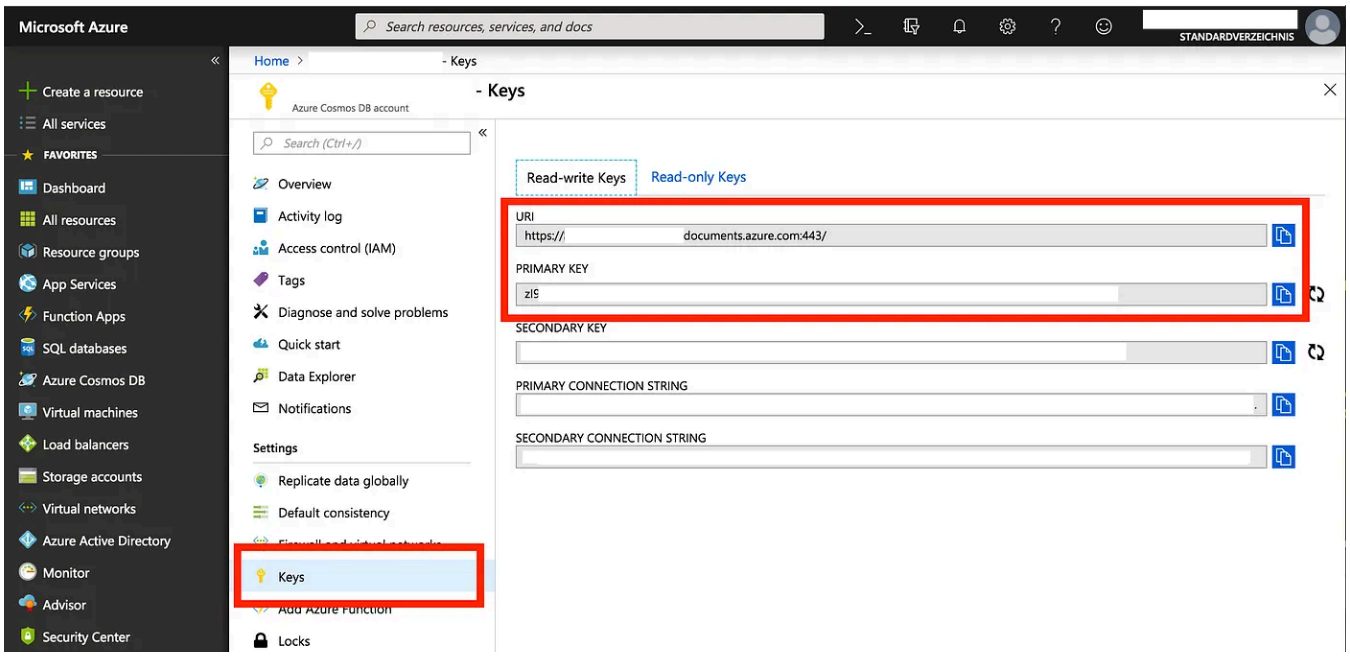


Add Graph

* Database id: [✓ Create new] Use existing
my-graph-db
Provision database throughput
* Graph Id: people
* Storage capacity: [✓ Fixed (10 GB)] Unlimited
* Throughput (400 - 10.000 RU/s): 400
Estimated spend (USD): $0.032 hourly / $0.77 daily.
Choose unlimited storage capacity for more than 10.000 RU/s.

Creating a new Cosmos DB Graph Database

After creating the database, you can switch to the “Keys” section and grab both the URI and primary key values. We will need both properties in the next section to establish a connection via C#.

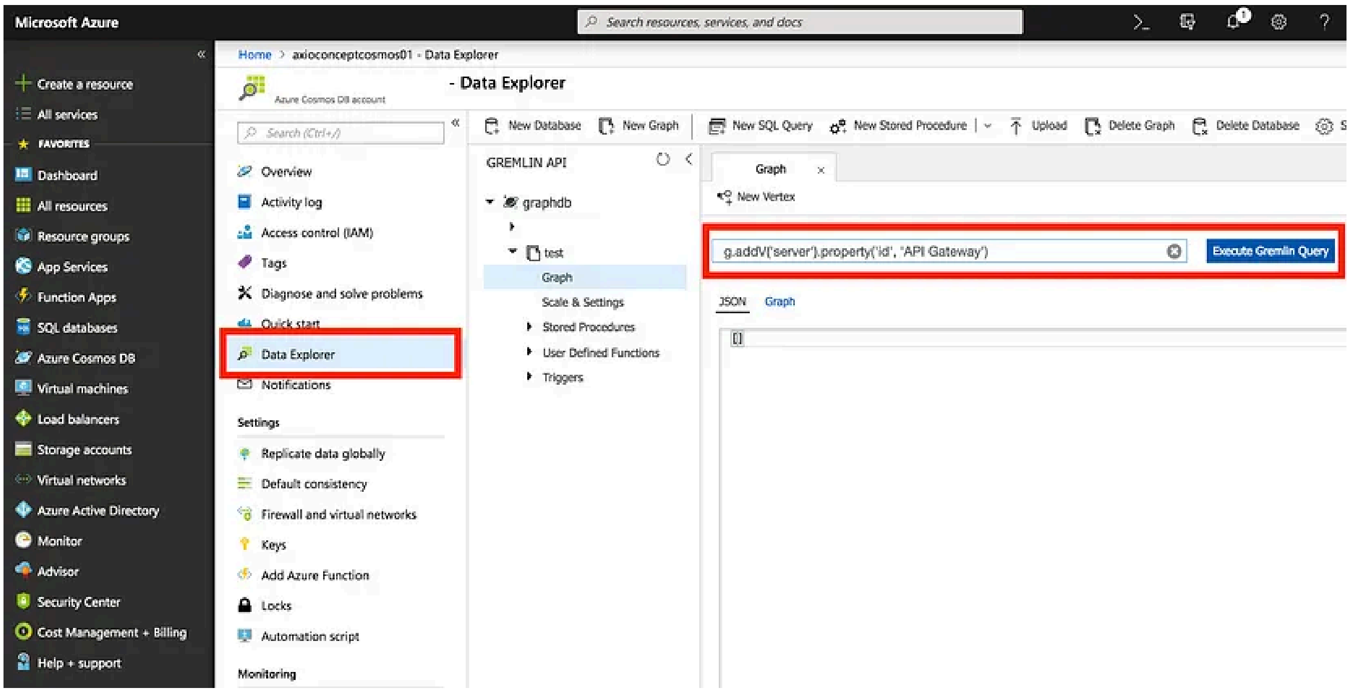


Getting the Cosmos DB Read-Write Key needed by the C# SDK

First Steps with Gremlin

Gremlin is a query language specifically designed for handling graph data structures. We can get familiar with some basic queries using the Data Cosmos DB Data Explorer. Go to the Data Explorer, select your previously created Graph Database and open the associated Graph. You will see an input box that allows you to run Gremlin queries.

Run the query `g.addV('server').property('id', 'API Gateway')` , as shown in the screenshot below:



```
g.addV('server').property('id', 'API Gateway')
```

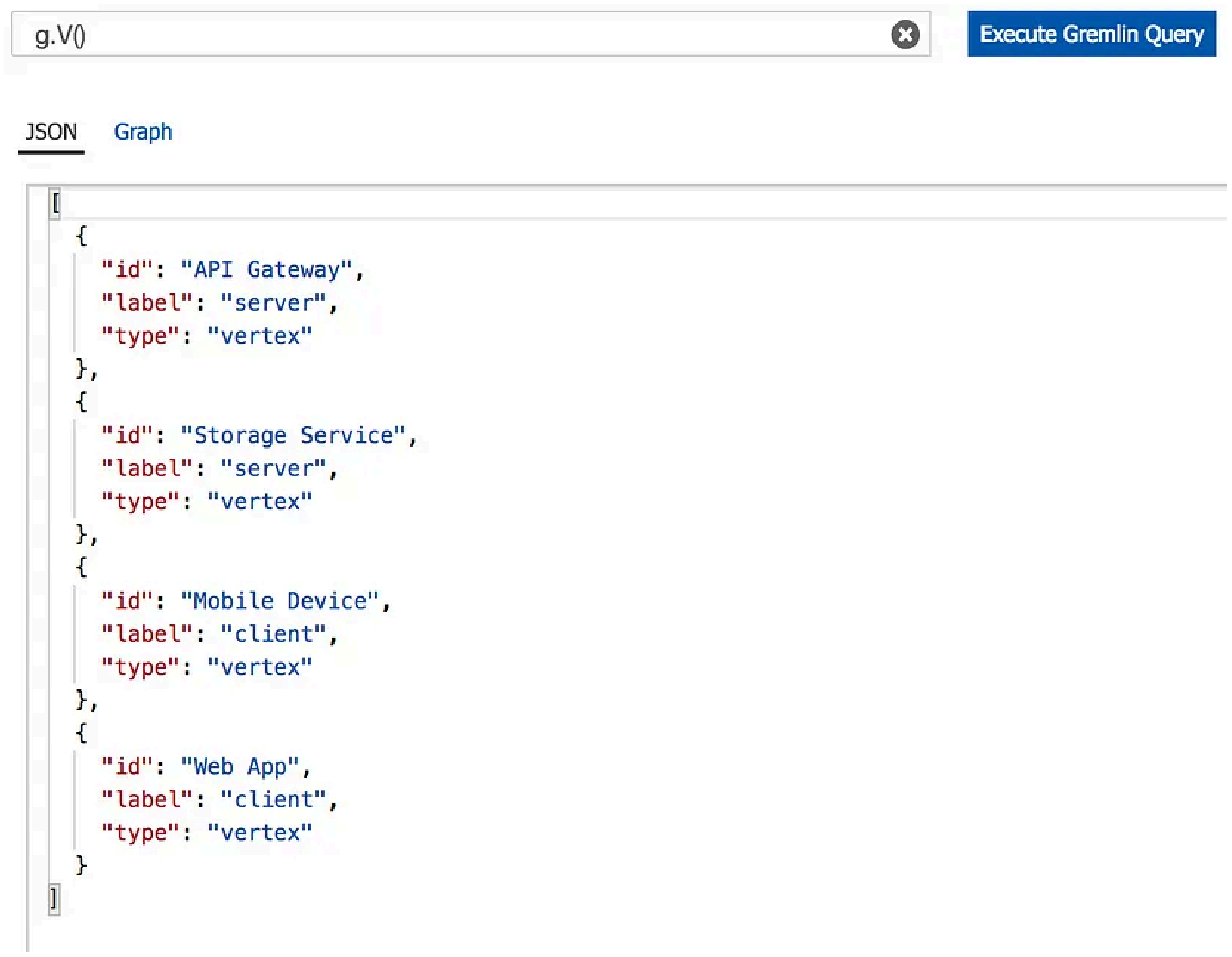
```
[]
```

Running Gremlin queries in the Azure Cosmos DB Data Explorer

The query will add a new vertex to the graph. The parameter of `addV` specifies the label of the vertex, which denotes its type. Let's add some more vertices by executing the following queries:

```
g.addV('server').property('id', 'Storage Service')
g.addV('client').property('id', 'Mobile Device')
g.addV('client').property('id', 'Web App')
```

By executing `g.V()` , we can retrieve all the vertices of the graph. The Data Explorer allows us to explore the query result either as JSON or as a graph visualization. We can see the following JSON result:



```
[
  {
    "id": "API Gateway",
    "label": "server",
    "type": "vertex"
  },
  {
    "id": "Storage Service",
    "label": "server",
    "type": "vertex"
  },
  {
    "id": "Mobile Device",
    "label": "client",
    "type": "vertex"
  },
  {
    "id": "Web App",
    "label": "client",
    "type": "vertex"
  }
]
```

A query result shown by the Data Explorer

Of course, we can query vertices by label and property values. The following query retrieves all server vertices:

```
g.V().hasLabel('server')
```

And the following two queries both retrieve the vertex that has the ID “Web App”:

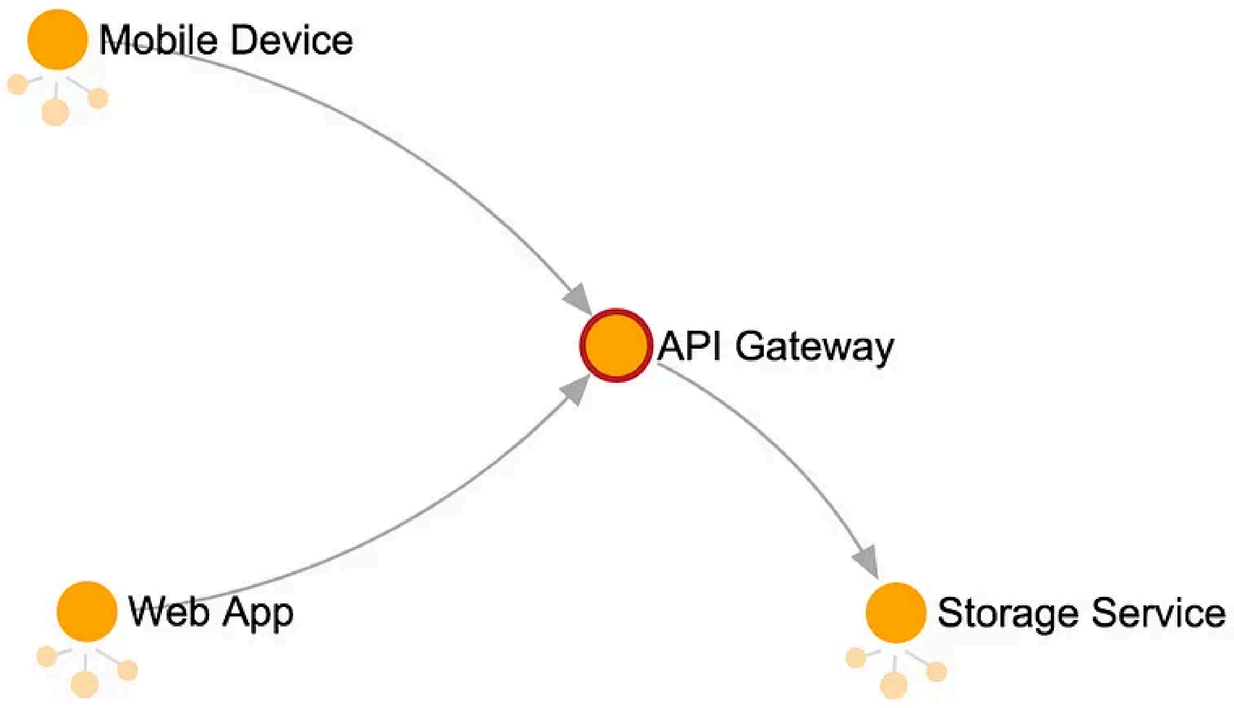
```
g.V().has('id', 'Web App')
g.V('Web App')
```

By adding some edges, we can make our graph more interesting:

```
g.V('Web App').addE('connects').to(g.V('API Gateway'))
g.V('Mobile Device').addE('connects').to(g.V('API Gateway'))
```

```
g.V('API Gateway').addE('connects').to(g.V('Storage Service'))
```

This will connect both client vertices to the API Gateway and the API Gateway to the Storage Service. By executing `g.V('API Gateway')` , we can examine the result:



Resulting Graph

Just like the vertices, edges also have a label, denoting their type. In our case, all edges are labeled “connects”. We can retrieve these edges by executing `g.E().hasLabel('connects')` .

Exercise 1:
Experiment with the following queries using the Data Explorer and describe the result:

```
A) g.V('API Gateway').inE()
B) g.V('API Gateway').inE().outV()
C) g.V('API Gateway').bothE()
D) g.V('API Gateway').inE().hasLabel('connects').count()
E) g.V('Web App').outE().inV().id()
```

Answer:

```
A) Retrieves incoming edges of the API Gateway

B) Retrieves the source vertices of the incoming edges of the API
Gateway

C) Retrieves both incoming and outgoing edges of the API Gateway

D) Counts the incoming 'connects'-edges of the API Gateway

E) Traverses from the Web App to the target vertices and get their
ID
```

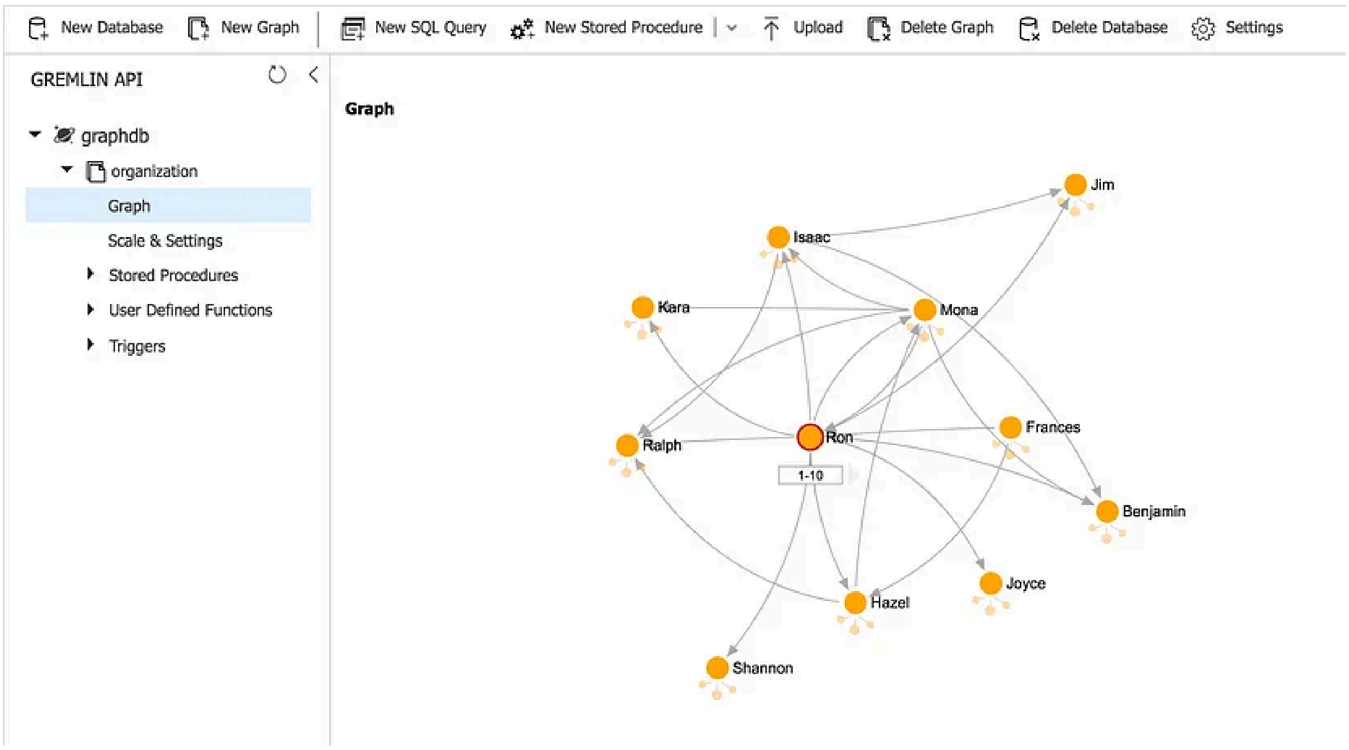
After the next section, we will take a look at some more advanced Gremlin queries. But before, we might need some more complex data to play with.

Graph Construction using Gremlin.Net

Gremlin.Net is an implementation of the Gremlin language based on .NET Standard. It offers both an API for executing Gremlin query strings, as well as a fluent API that allows a typesafe approach towards graph queries. However, at the time of writing, Cosmos DB does not yet support the fluent API due to a missing bytecode stream implementation. There is a related discussion on Github: https://github.com/Azure/azure-cosmosdb-dotnet/issues/439

There is an open source project, Gremlin.Net.CosmosDb, that aims to provide fluent API queries by serializing them into a Gremlin string representation. You can get more information on the Github project page: https://github.com/evo-terren/Gremlin.Net.CosmosDb

For starters, we will send some plain Gremlin query strings to build and upload a graph. In the following example, we will construct a sociogram and upload it to Azure Cosmos DB. The resulting graph will look like this:



Vertices and Edges visualized by the Cosmos DB Data Explorer

You can grab the example on Github: https://github.com/marco-bue/azure-cosmos-gremlin