Mihail Mateev

Senior Solution Architect @EPAM

Microsoft MVP – Azure, Data Platform

Microsoft Regional Director

PASS Regional Mentor

Dealing with CosmosDB

BIG Thanks to SQL Sat Denmark sponsors

GOLD

kapacity Business Analytics

TIMEXTENDER

Quest

Microsoft

SILVER

NEXTAGENDA
BUSINESS INTELLIGENCE LØSNINGER

BRONZE

ORANGEMAN

accobat
Know your business

Raffle and goodbye Beer

Remember to **visit the sponsors**, stay for **the raffle** and **goodbye beers** ☺

Join our sponsors for a lunch break session in :
cust 0.01 and **cust 1.06**

We hope you'll all have a great Saturday.

Regis, Kenneth

About me

Mihail Mateev is a Technical Consultant,
Community enthusiast, PASS RM for CEE,
PASS and chapter lead, Microsoft Regional Director,
Microsoft MVP - Microsoft Azure,
Ph.D. on Cloud Computing

Senior Solutions Architect at EPAM Systems

Mihail works in various areas related to Microsoft technologies : .Net Framework, IoT Suite,
Data Platform and Microsoft Azure

About this talk

Agenda:

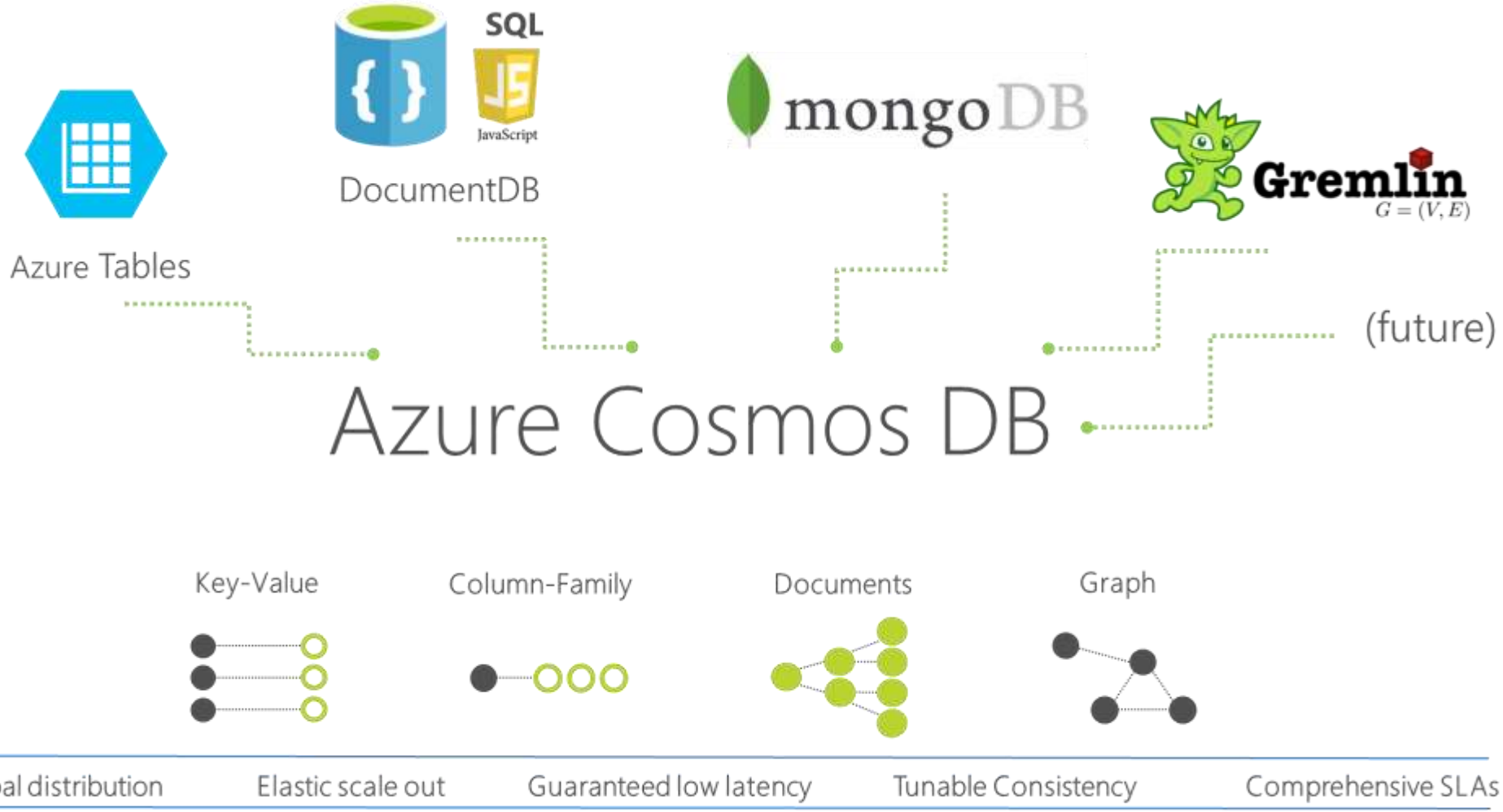
- What is Azure CosmosDB ?
- Azure CosmosDB Resources
- Developing Against Azure CosmosDB
- Partitioning data in CosmosDB
- Querying CosmosDB
- Azure CosmosDB Performance
- Azure CosmosDB Real Life Cases
- CosmosDB Last Updates

What is CosmosDB ?

What is Azure CosmosDB

The CosmosDB launched a new Azure cloud database as a service replacing Document DB which was an earlier choice of the company.

What is Azure CosmosDB



A multi-model, globally-distributed database service

What is Azure CosmosDB

Global Distribution

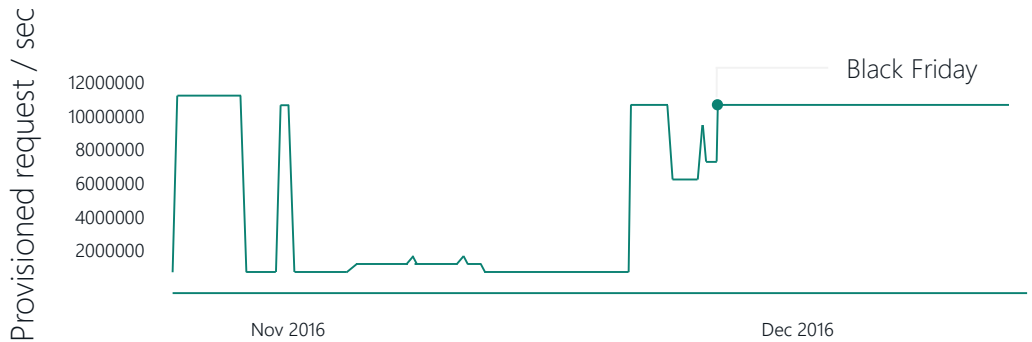
Worldwide presence

Automatic multi-region replication

Multi-homing APIs

Manual and automatic failovers

What is Azure CosmosDB



Elastically Scale-out

Partition management is automatically taken care of for you

Independently scale storage and throughput

Scale storage from Gigabytes to Petabytes

Scale throughput from 100's to 100,000,000's of requests/second

Dial up/down throughput and provision only what is needed

What is Azure CosmosDB

	Reads (1KB)	Indexed Writes (1KB)
P50	<2ms	<6ms
P99	<10ms	<15ms

Guaranteed low latency

Globally distributed with requests served from local region

Write optimized, latch-free database

Automatic Indexing

What is Azure CosmosDB

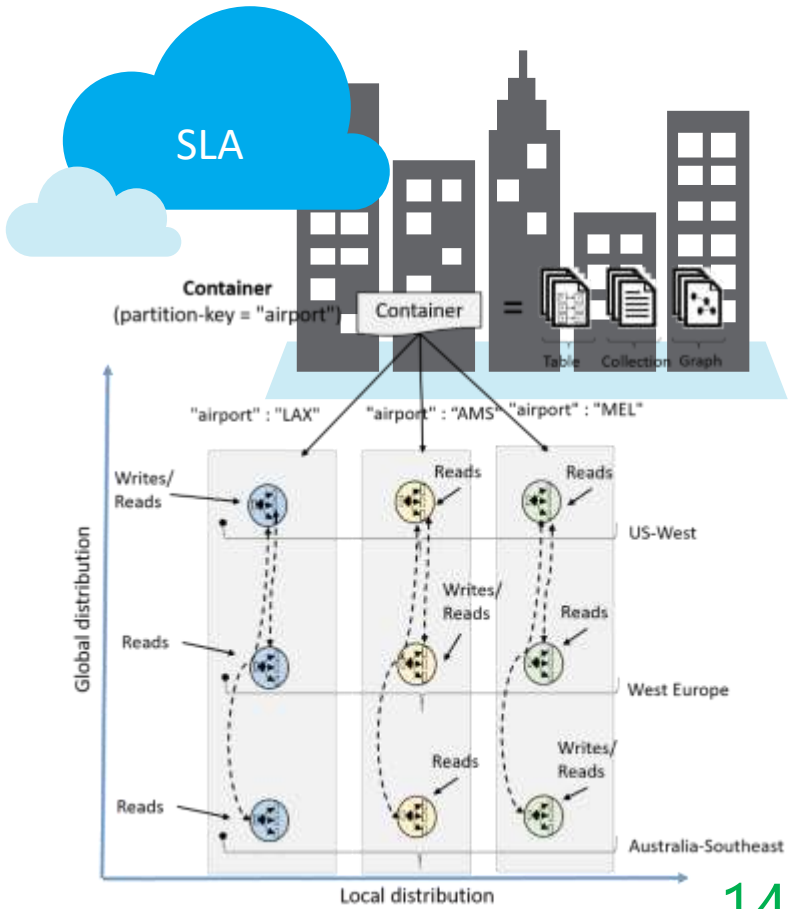
Five Consistency Models

Helps navigate Brewer's CAP theorem

Intuitive Programming
- Tunable well-defined consistency levels
- Override on per-request basis

Clear PACELC tradeoffs
- Partition – Availability vs Consistency
- Else – Latency vs Consistency

What is Azure CosmosDB



Comprehensive SLAs

99.99% availability

Durable quorum committed writes

Latency, consistency, and throughput also covered by financially backed SLAs

Made possible with highly-redundant architecture

What is Azure CosmosDB

The CosmosDB launched a new Azure cloud database as a service replacing Document DB which was an earlier choice of the company.

What is Azure CosmosDB

Starting with is easy

What is Azure CosmosDB

The CosmosDB is not based on .Net Framework

CosmosDB vs DocumentDB

CosmosDB vs DocumentDB

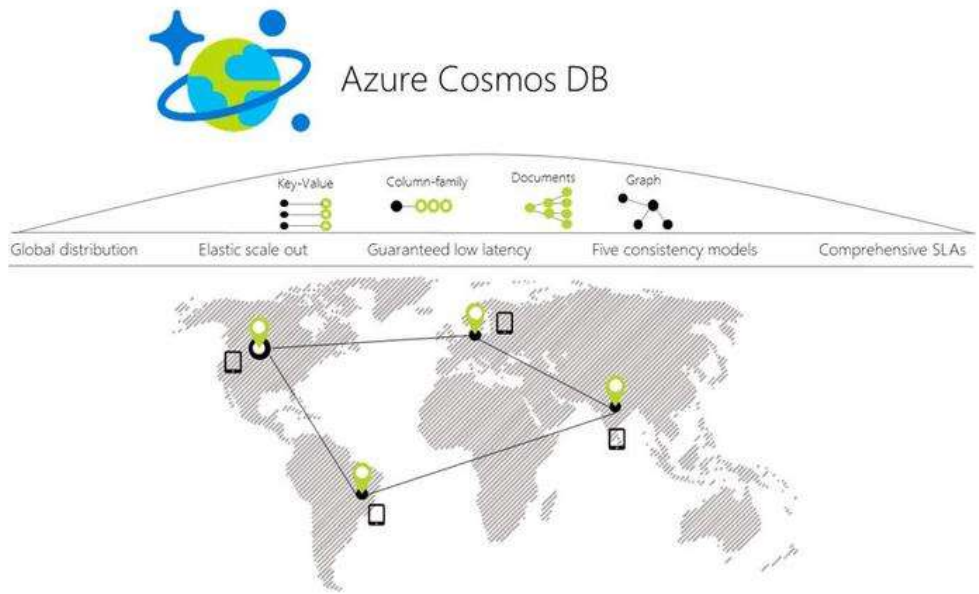
- Cosmos is a piece of distributed database technology originally built for Microsoft's internal use. Hence, it is the data backend for a whole lot of Microsoft's own services.
- DocumentDB, introduced in 2014, was a slice of its features – namely, a database designed for storing JSON documents.

CosmosDB vs DocumentDB

- DocumentDB also featured a SQL-like query syntax that made it easy to manage. Later on, it added support for MongoDB APIs, making it even useful even for software not specifically built for Azure.

CosmosDB vs DocumentDB

- Azure Cosmos DB is a superset of Microsoft's existing NoSQL DocumentDB service.
- Microsoft is transitioning all existing DocumentDB customers and their data to Azure Cosmos DB for no additional charge.



CosmosDB vs DocumentDB

- Cosmos DB is “a major leap forward” from what DocumentDB was able to offer.
- DocumentDB only offered a subset of the capabilities of what is now Cosmos DB.
- While DocumentDB was essentially a store for JSON data, Cosmos DB extends the idea of an index-free database system and adds support for various new data types that allows Cosmos DB enough flexibility to work as a graph database or key-value database, for example. And for those who are looking to store more traditional columnar relational data, Cosmos DB will also offer support for those.

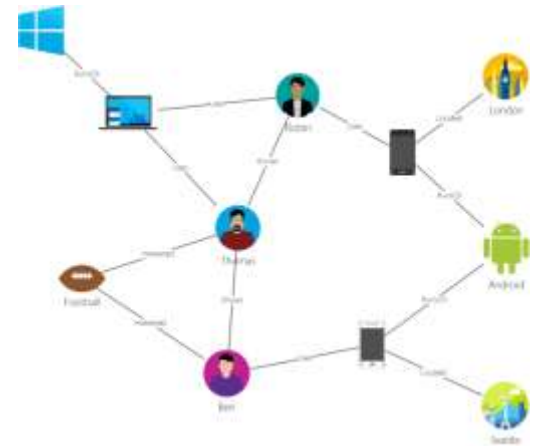
CosmosDB Data Formats

CosmosDB Data Formats

- The DocumentDB data atom is a JSON document. Cosmos DB broadens the support to include two additional data shapes: graphs and tables.
- Support for MongoDB APIs makes possible to migrate solutions, working with MongoDB to CosmosDB / DocumentDB

CosmosDB Data Formats

- Graphs are composed of nodes combined with edges – a very real-wordly representation of data. They are queried using a common open source query language called Gremlin.



- The query syntax is geared at navigating graphs – you could say e.g. .has('person', 'name', 'Thomas').outE('Knows') to find people who Thomas knows.

CosmosDB Data Formats

- Tables, on the other hand, provide a non-relational tabular data model. While the Cosmos DB is a separate data engine, the API and the data model used is the same as it is with Azure Table Storage.
- You can port your current Table Storage driven apps to Cosmos DB with relative ease – just change the connection string (and migrate your data, which is another discussion entirely).

CosmosDB Key Features

Document DB

DocumentDB key capabilities and benefits

- Elastically scalable throughput and storage
- Multi-region replication
- Ad hoc queries with familiar SQL syntax
- JavaScript execution within the database
- Tunable consistency levels
- Fully managed
- Open by design
- Automatic indexing

DocumentDB Storage

Elastic SSD:

- Makes collection truly elastic
- Add/Remove documents grows/shrinks collection
- Tested with real-world clients from gigabytes to terabytes

DocumentDB Indexing

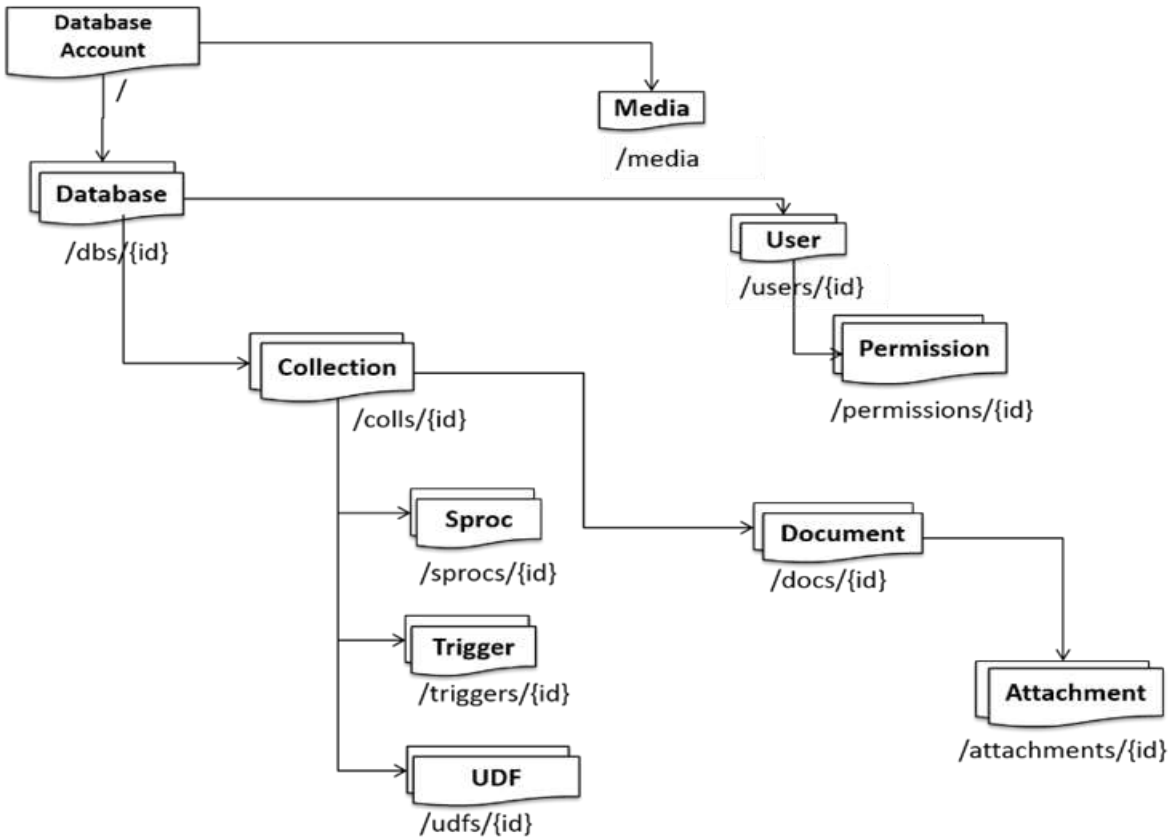
Automatic Indexing

- Indexing on by default
- Can optimize for performance and storage tradeoffs
- Index only specific paths in your document
- Synchronous indexing at write time by default
- Can be Asynchronous for boosted write performance
- Eventually consistent

Document DB Resources

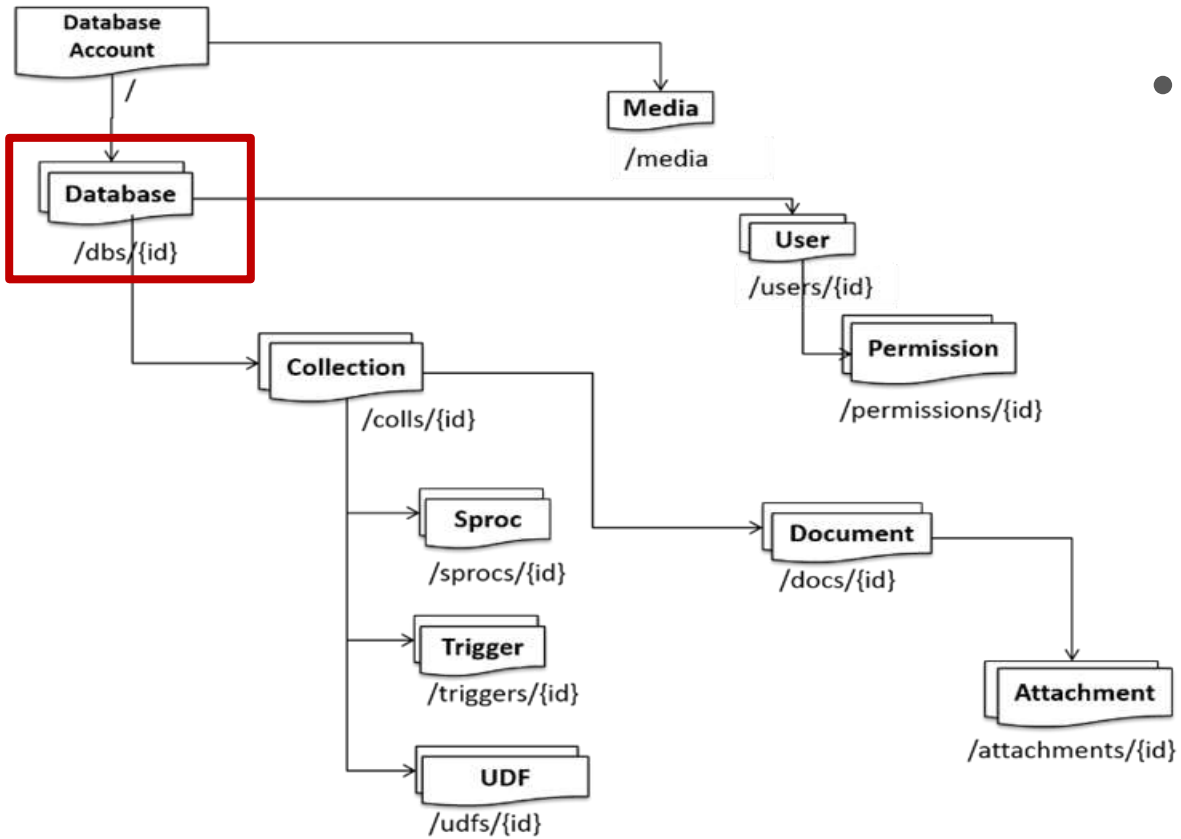
DocumentDB Resources

Understanding the DocumentDB structure:



DocumentDB Resources

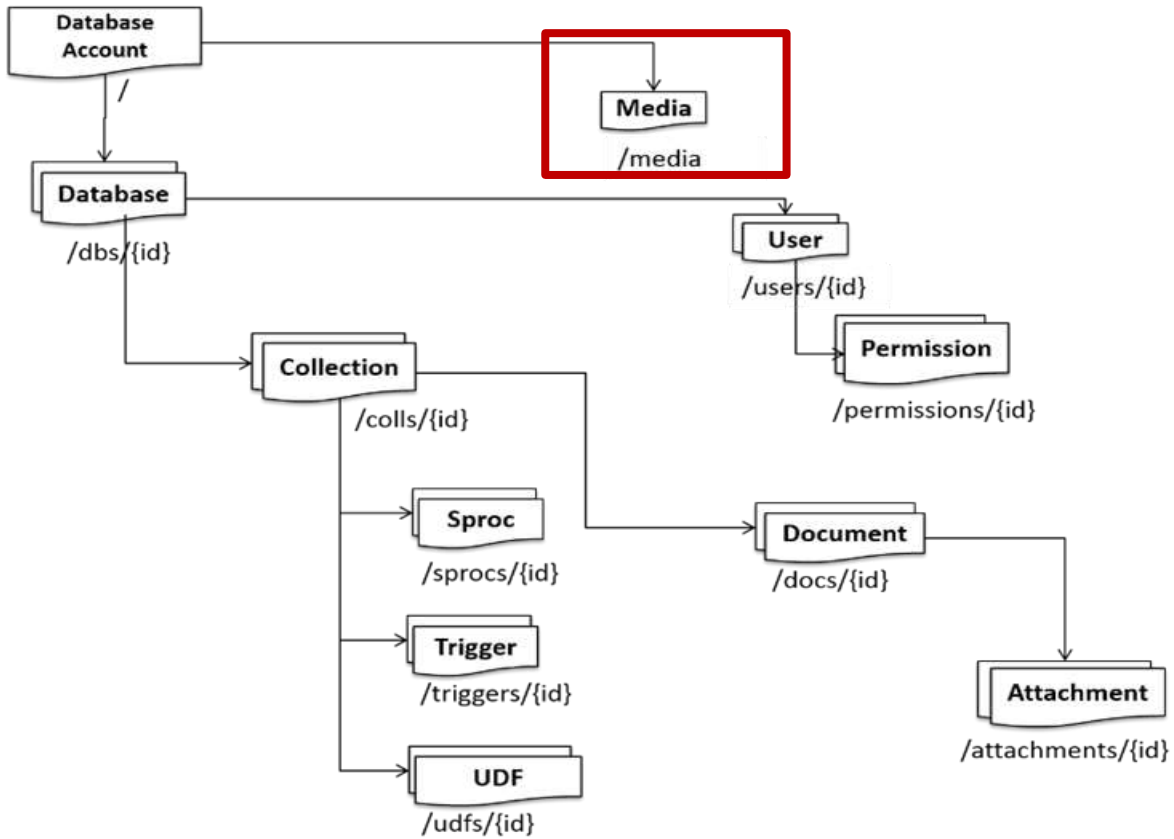
Structure: Database :



- The container that houses your data
- /dbs/{id} is not your ID
  - Hash known as a “Self Link”

DocumentDB Resources

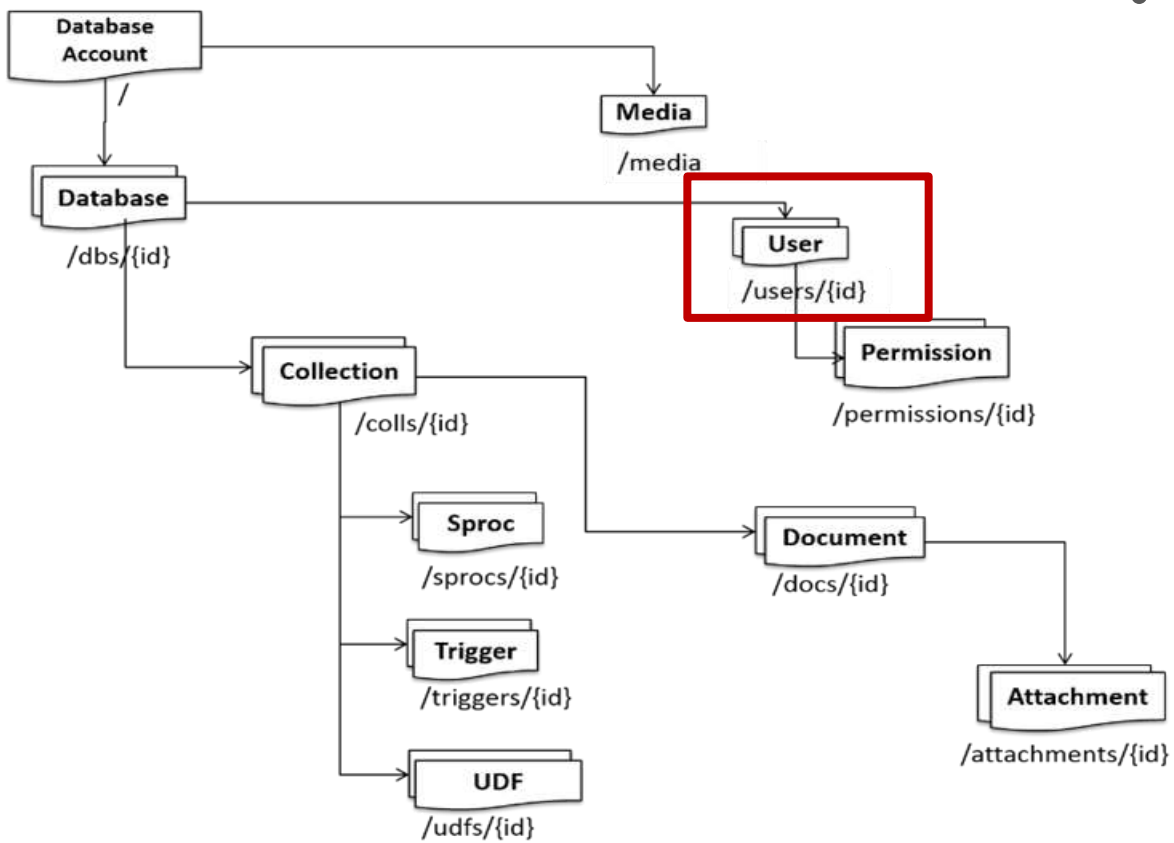
Structure: Media:



- Video
- Audio
- Blob
- Etc.

DocumentDB Resources

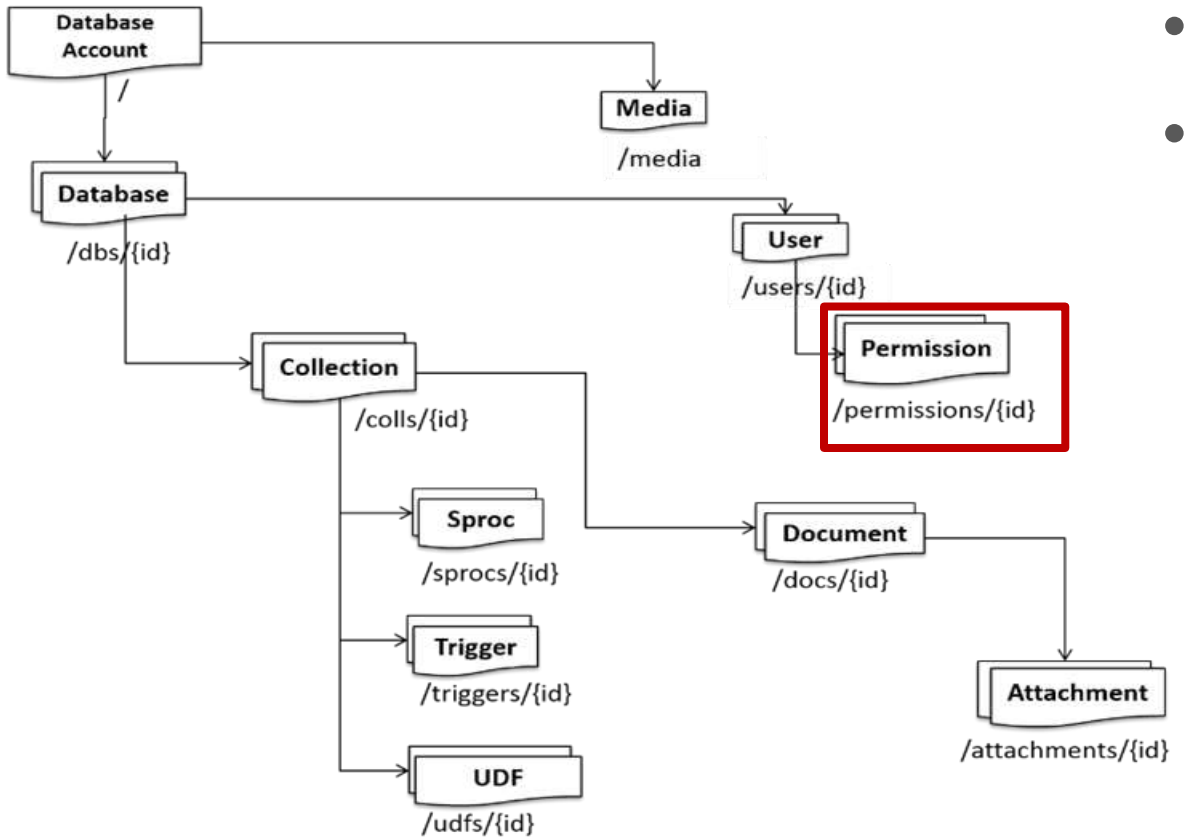
Structure: User:



- Invite in an existing azure account
- Allows you to set permissions on each concept of the database

DocumentDB Resources

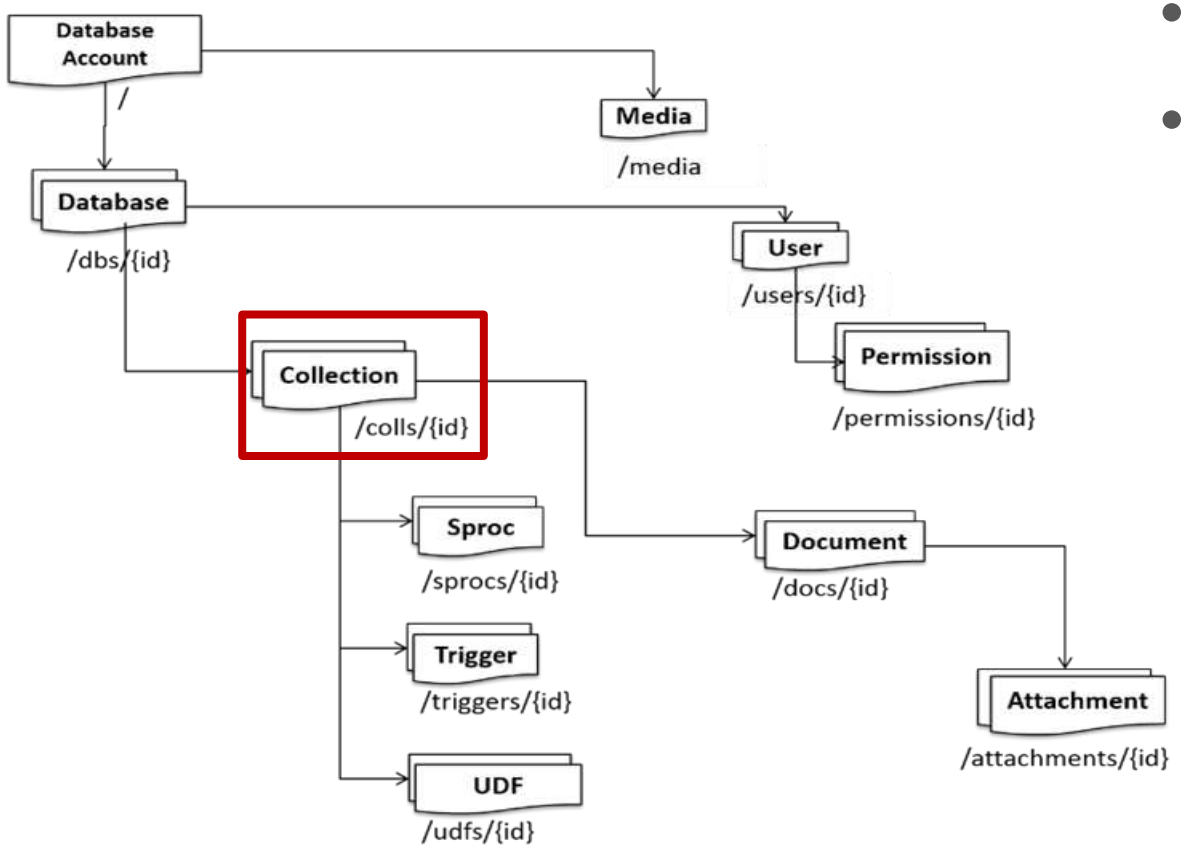
Structure: Permission:



- Authorization token
- Associated with a user
- Grants access to a given resource

DocumentDB Resources

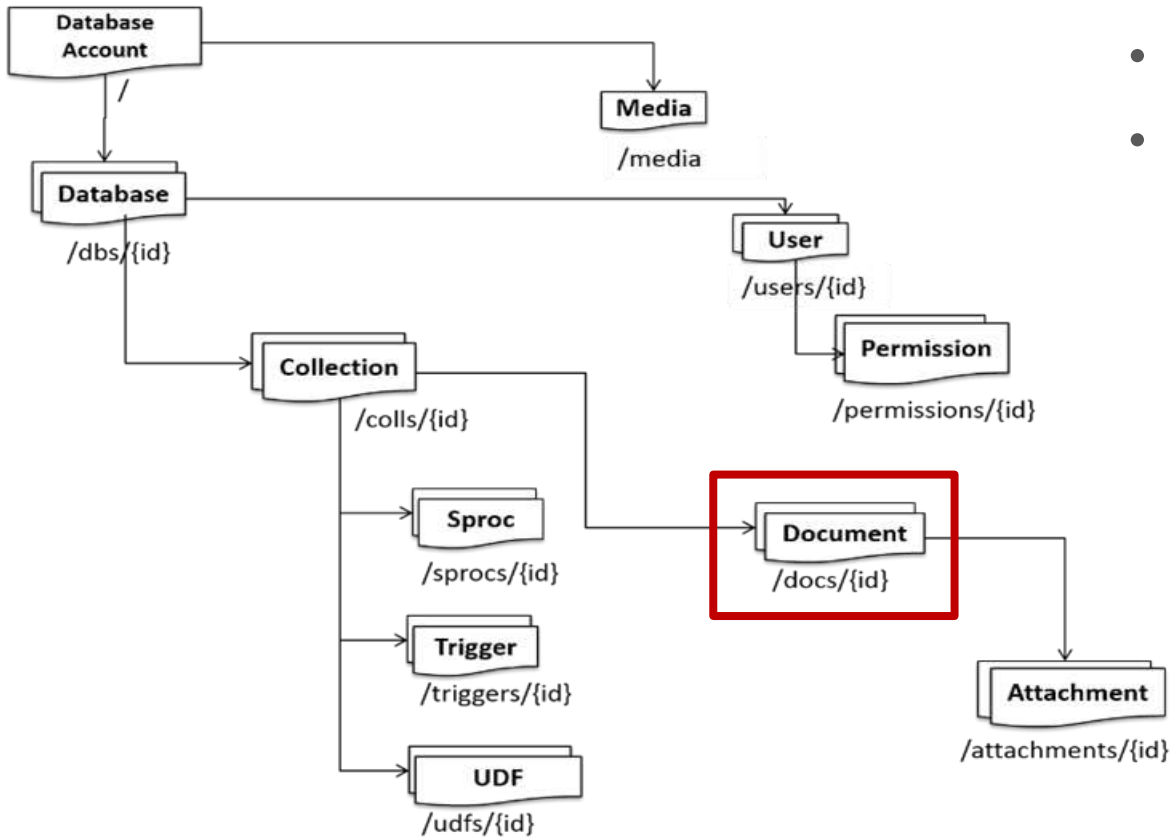
Structure: Collection:



- Most like a “table”
- Structure is not defined
- Dynamic shapes based on what
  you put in it

DocumentDB Resources

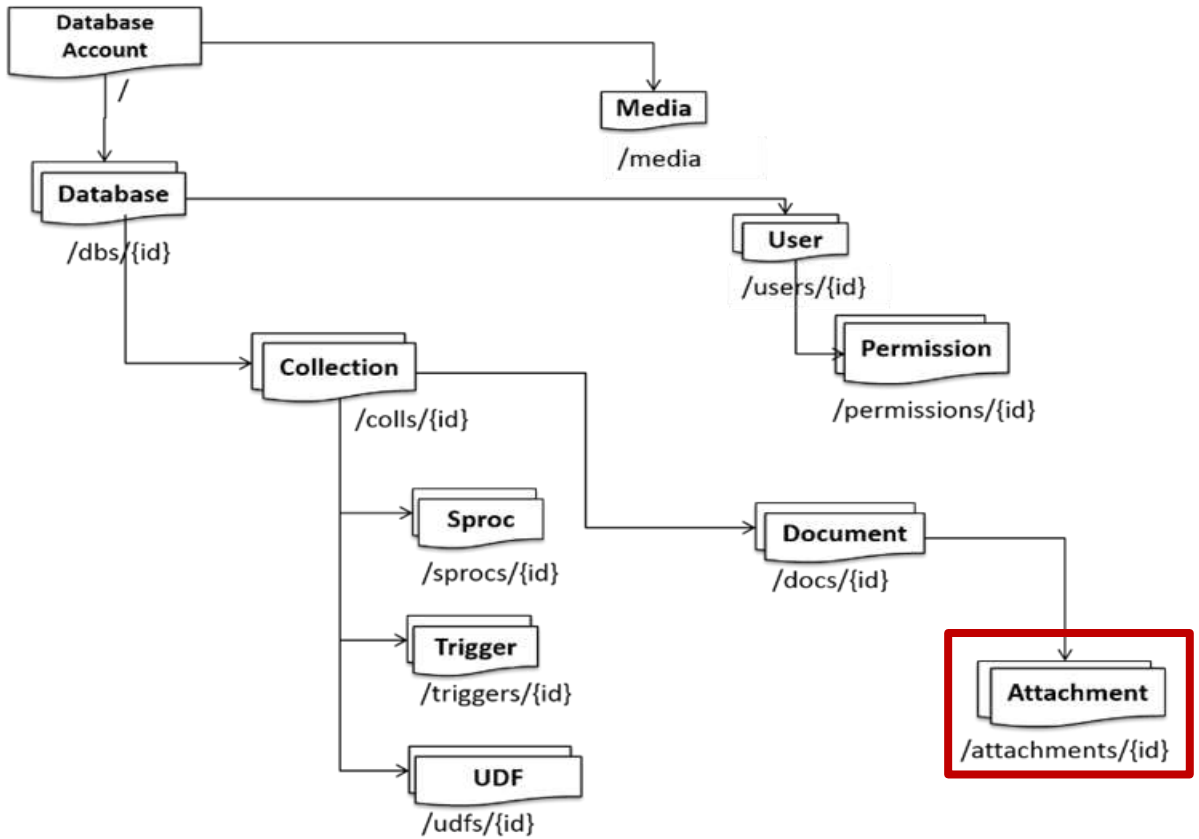
Structure: Document:



- A blob of JSON representing your data
- Can be a deeply nested shape
- No specialty types
- No specific encoding types

DocumentDB Resources

Structure: Attachment:



- Think media – at the document level!

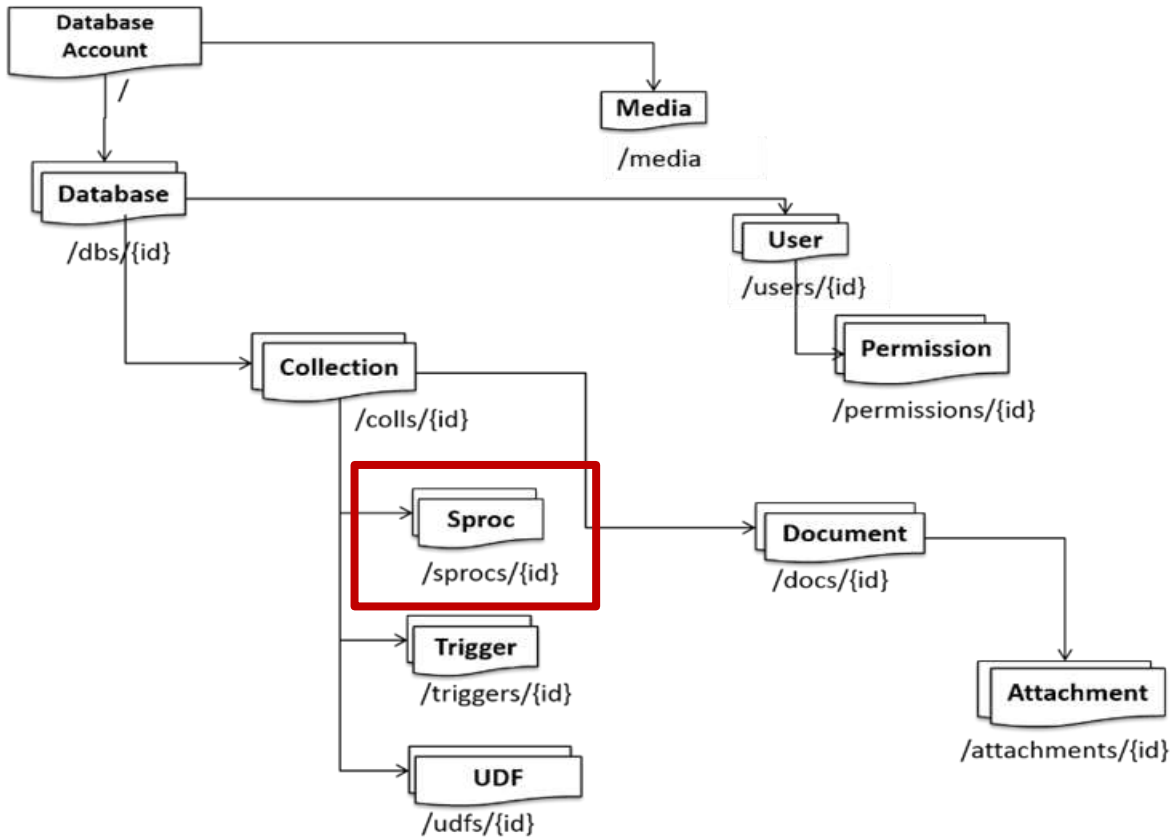
The maximum size for a document and it's attachment in CosmosDB now is 2 MB.

In DocumentDB the maximum size of document and it’s attachment was 512KB

Media size – 2GB

DocumentDB Resources

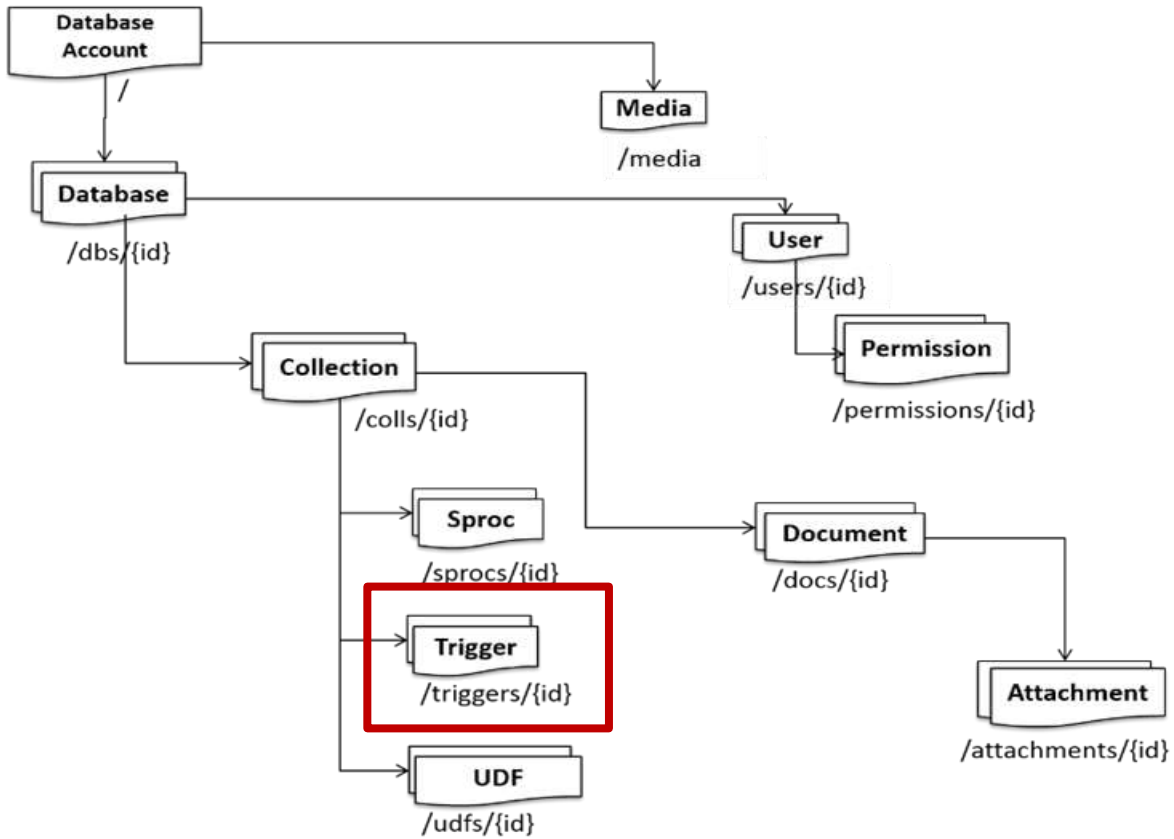
Structure: Stored Procedure:



- Written in JavaScript!
- Is transactional
- Executed by the database engine
- Can live in the store
- Can be sent over the wire

DocumentDB Resources

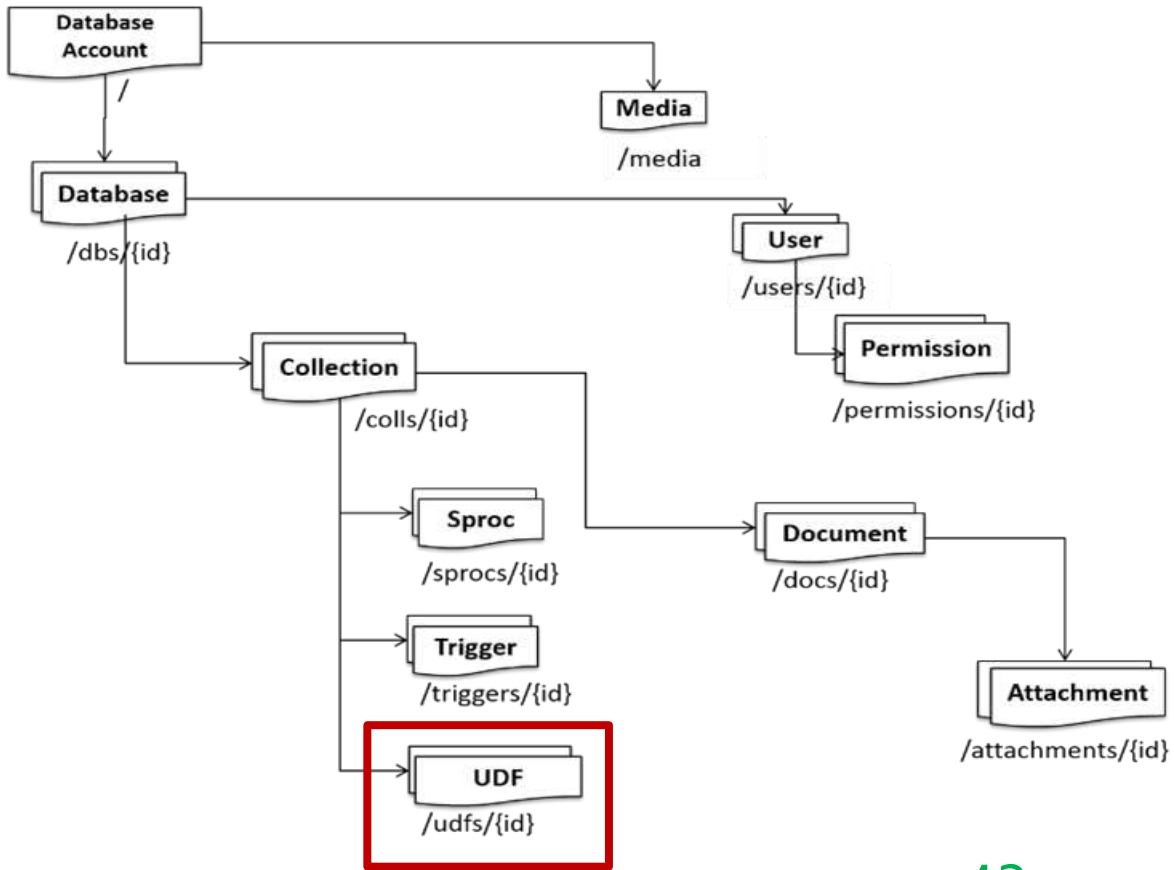
Structure: Triggers:



- Can be Pre or Post (before or after)
- Can operate on the following actions
  - Create
  - Replace
  - Delete
  - All
- Also written in javascript!

DocumentDB Resources

Structure: UDF:



- Can only be ran on a query
- Modifies the result of a given query
- mathSqrt()

DocumentDB Resources

Indexing: The DocumentDB indexing subsystem is designed to support

- Rich hierarchical and relational queries without any schema or index definitions
- Consistent query results while handling a sustained volume of writes
- Storage efficiency.
- Multi-tenancy

DocumentDB Resources

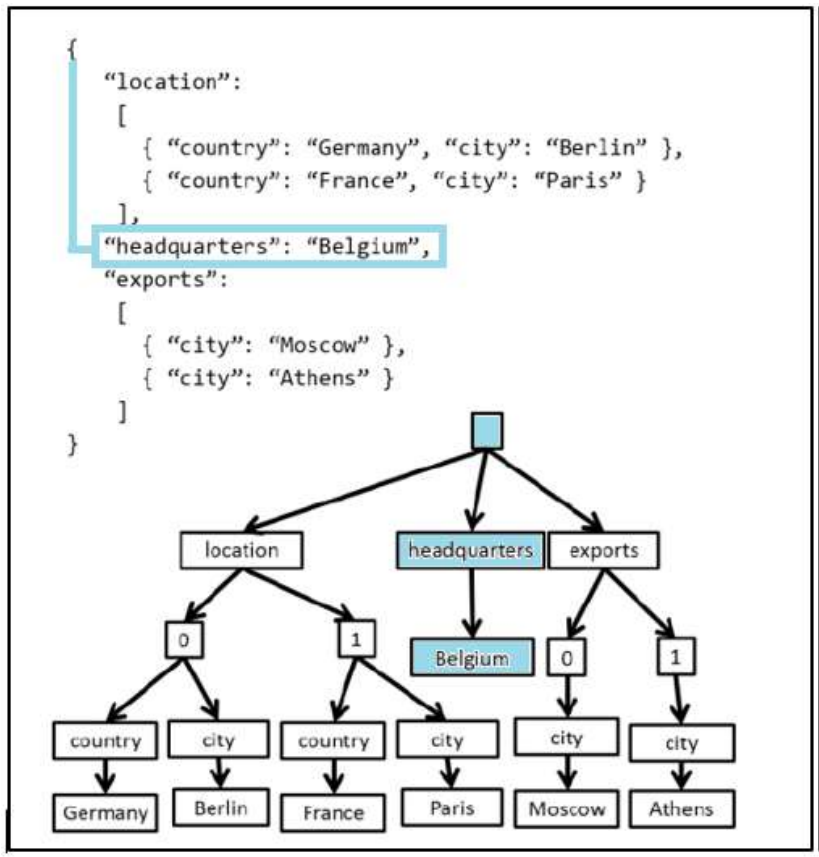
Indexing: How DocumentDB indexing works

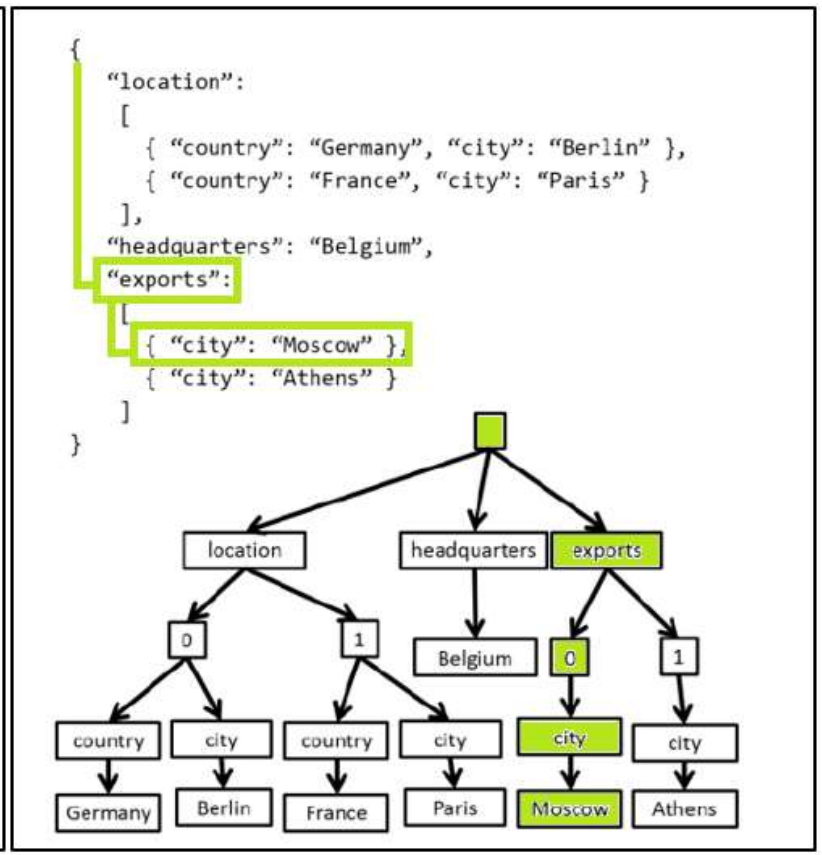
- **DocumentDB** takes advantage of the fact that **JSON** grammar allows documents to be represented as trees:

  JSON array {"exports": [{"city": “Moscow"}, {"city": Athens"}]} correspond to the paths /"exports"/0/"city"/"Moscow" and /"exports"/1/"city"/"Athens".

DocumentDB Resources

Indexing: How DocumentDB indexing works





DocumentDB Resources

Indexing: How DocumentDB indexing works

Path	Description/use case
/	Default path for collection. Recursive and applies to whole document tree.
/"prop"/?	Index path required to serve queries like the following (with Hash or Range types respectively): **SELECT * FROM collection c WHERE c.prop = "value"** **SELECT * FROM collection c WHERE c.prop > 5**
/"prop"/*	Index path for all paths under the specified label. Works with the following queries SELECT * FROM collection c WHERE c.prop = "value" SELECT * FROM collection c WHERE c.prop.subprop > 5 **SELECT * FROM collection c WHERE c.prop.subprop.nextprop = "value"**

DocumentDB Resources

Indexing: How DocumentDB indexing works

Path	Description/use case
/"props"/[]/?	Index path required to serve iteration and JOIN queries against arrays of scalars like ["a", "b", "c"]: **SELECT tag FROM tag IN collection.props WHERE tag = "value"** **SELECT tag FROM collection c JOIN tag IN c.props WHERE tag > 5**
/"props"/[]/"subprop"/?	Index path required to serve iteration and JOIN queries against arrays of objects like [{subprop: "a"}, {subprop: "b"}]: **SELECT tag FROM tag IN collection.props WHERE tag.subprop = "value"** **SELECT tag FROM collection c JOIN tag IN c.props WHERE tag.subprop = "value"**
/"prop"/"subprop"/	Index path used during query execution to prune documents that do not have the specified path.
/"prop"/"subprop"/?	Index path required to serve queries (with Hash or Range types respectively): SELECT * FROM collection c WHERE c.prop.subprop = "value" **SELECT * FROM collection c WHERE c.prop.subprop > 5**

DocumentDB Resources

Structure: UDF:

```
var filterQuery = "SELECT mathSqrt(r.Age) AS sqrtAge FROM root r WHERE r.FirstName='John'";
client.queryDocuments(collection._self, filterQuery).toArrayAsync();
    .then(function(queryResponse) {
        var queryResponseDocuments = queryResponse.feed;
    }, function(error) {
        console.log("Error");
    });
```

DocumentDB Resources

System vs. user defined resources

Property	User settable or system generated?	Purpose
_rid	System generated	System generated, unique and hierarchical identifier of the resource.
_etag	System generated	etag of the resource required for optimistic concurrency control.
_ts	System generated	Last updated timestamp of the resource.
_self	System generated	Unique addressable URI of the resource.
id	User settable	User defined unique name of the resource.

DocumentDB Resources

Addressing a resource

Value of the _self	Description
/dbs	Feed of databases under a database account.
/dbs/{_rid-db}	Database with the unique id property with the value {_rid-db}.
/dbs/{_rid-db}/colls/	Feed of collections under a database.
/dbs/{_rid-db}/colls/{_rid-coll}	Collection with the unique id property with the value {_rid-coll}.
/dbs/{_rid-db}/users/	Feed of users under a database.
/dbs/{_rid-db}/users/{_rid-user}	User with the unique id property with the value {_rid-user}.
/dbs/{_rid-db}/users/{_rid-user}/permissions	Feed of permissions under a database.
/dbs/{_rid-db}/users/{_rid-user}/permissions/{_rid-permission}	Permission with the unique id property with the value {_rid-permission}.

DocumentDB Resources

Performance levels in DocumentDB:

- Each DocumentDB collection created under a Standard account is provisioned with an associated performance level.
- Each performance level has an associated request unit* (RU) rate limit.

Collection performance level	Reserved throughput
S1	250 RU/sec
S2	1000 RU/sec
S3	2500 RU/sec

DocumentDB Resources

Performance levels in DocumentDB:

What is a *request unit (RU)

* A single **request unit** represents the processing capacity required to read (via self link) a single 1KB JSON document consisting of 10 unique property values

DocumentDB Resources

Performance levels changes in DocumentDB:

- The S1, S2, and S3 performance levels discussed in this article are being retired and are no longer available for new DocumentDB API accounts.
- These performance levels were migrated to single partition collections on July 21st, 2017.

DocumentDB Resources

Single partition collections and partitioned collections, compared to the S1, S2, S3 performance levels

	Partitioned collection	Single partition collection	S1	S2	S3
Maximum throughput	Unlimited	10K RU/s	250 RU/s	1 K RU/s	2.5 K RU/s
Minimum throughput	2.5K RU/s	400 RU/s	250 RU/s	1 K RU/s	2.5 K RU/s
Maximum storage	Unlimited	10 GB	10 GB	10 GB	10 GB
Price	Throughput: $6 / 100 RU/s Storage: $0.25/GB	Throughput: $6 / 100 RU/s Storage: $0.25/GB	$25 USD	$50 USD	$100 USD

Partitioning in Azure CosmosDB

Partitioning in Cosmos DB

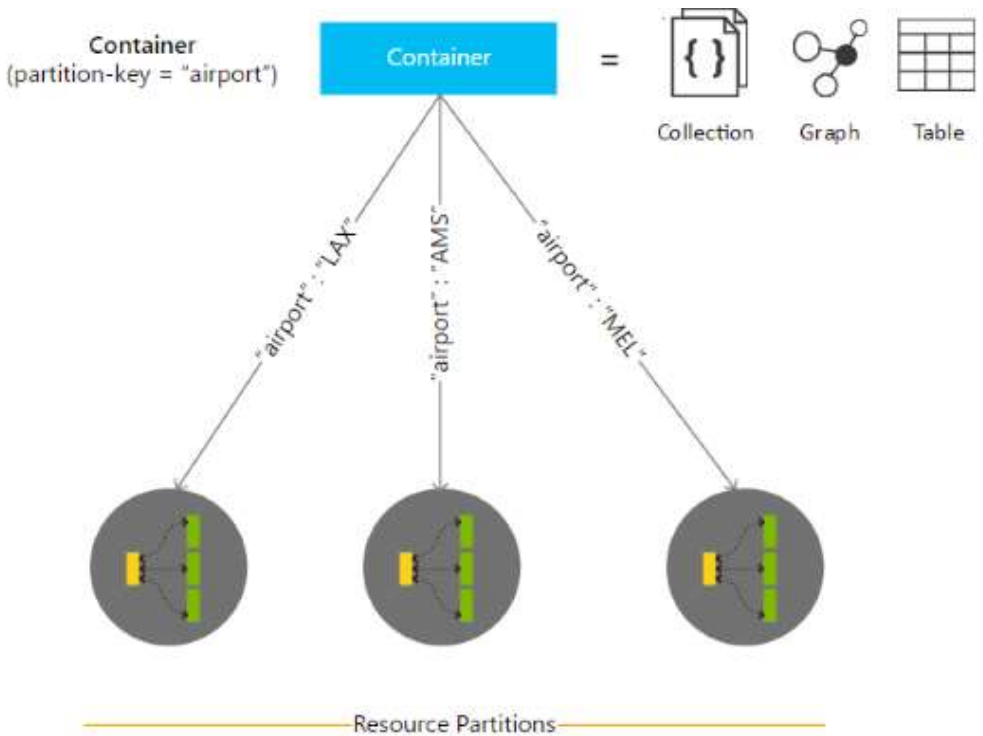
- Cosmos DB provides containers for storing data called collections (for document), graphs, or tables.
- Containers are logical resources and can span one or more physical partitions or servers.
- Every partition in Cosmos DB has a fixed amount of SSD-backed storage associated with it, and is replicated for high availability.

Partitioning in Cosmos DB

- Partition management is fully managed by Azure Cosmos DB, and you do not have to write complex code or manage your partitions. Cosmos DB containers are unlimited in terms of storage and throughput.

Partitioning in Cosmos DB

- Partitioning is transparent to your application.



Partitioning in Cosmos DB

- Each item must have a partition key and a row key, which uniquely identify it.
- The partition key acts as a logical partition for your data, and provides Cosmos DB with a natural boundary for distributing data across partitions

Partitioning in Cosmos DB

- You provision a Cosmos DB container with T requests/s throughput
- Behind the scenes, Cosmos DB provisions partitions needed to serve T requests/s. If T is higher than the maximum throughput per partition t, then Cosmos DB provisions N = T/t partitions
- Cosmos DB allocates the key space of partition key hashes evenly across the N partitions. So, each partition (physical partition) hosts 1-N partition key values (logical partitions)

Partitioning in Cosmos DB

- When a physical partition p reaches its storage limit, Cosmos DB seamlessly splits p into two new partitions p1 and p2 and distributes values corresponding to roughly half the keys to each of the partitions. This split operation is invisible to your application.
- Similarly, when you provision throughput higher than t*N throughput, Cosmos DB splits one or more of your partitions to support the higher throughput

Partitioning in Cosmos DB

- The semantics for partition keys

API	Partition Key	Row Key
DocumentDB	custom partition key path	fixed **id**
MongoDB	custom shard key	fixed **_id**
Graph	custom partition key property	fixed **id**
Table	fixed **PartitionKey**	fixed **RowKey**

Developing against CosmosDB (DocumentDB API)

Developing against DocumentDB API

- Net
- Node.js
- Java
- JavaScript
- Python

Developing against DocumentDB API

- DocumentDB .NET SDK
  - Connect to a DocumentDB account
  - References

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using System.Net;
using Newtonsoft.Json;
```

  - DocumentDB andpoint and access key

```
private static string EndpointUrl = "<your endpoint URI>";
private static string PrimaryKey = "<your key>";
```

Developing against DocumentDB API

- DocumentDB .NET SDK
  - Connect to a DocumentDB account
  - Create a task that cerates a client:

```
private static async Task GetStartedDemo()
{
        // Create a new instance of the DocumentClient.
        var client = new DocumentClient(new Uri(EndpointUrl),   PrimaryKey);
}
```

Querying CosmosDB (DocumentDB API)

Querying DocumentDB API

What is more important for a C# developer?

- Support for gateways and direct connectivity
- Async APIs for all operations.
- HTTP and TCP transports available.
- POCOs, inherited document types and dynamics.
- LINQ, LINQ, LINQ

Querying DocumentDB API

Document

```
{
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
      { "firstName": "Thomas" },
      { "firstName": "Mary Kay"}
    ],
    "children": [
      {
          "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,
          "pets": [{ "givenName": "Fluffy" }]
      }
    ],
    "address": { "state": "WA", "county": "King", "city": "seattle" },
    "isRegistered": true
}
```

Querying DocumentDB API

Create a document store

- Everything is done asynchronously!
- The ID of a new database is the friendly name

```
database = await GetClient().CreateDatabaseAsync(new
Database { Id = id });
```

Querying DocumentDB API

Creating a partitioned collection

- Creating a collection with 3,000 request units per second using the .NET SDK:

```
DocumentCollection myCollection = new DocumentCollection();
myCollection.Id = "coll";
myCollection.PartitionKey.Paths.Add("/deviceId"); await
client.CreateDocumentCollectionAsync(
UriFactory.CreateDatabaseUri("db"), myCollection, new
RequestOptions { OfferThroughput = 3000 });
```

Querying DocumentDB API

Creating a collection

```
this.client = new DocumentClient(new Uri(EndpointUrl),
PrimaryKey);
await this.client.CreateDatabaseIfNotExistsAsync
(new Database { Id = "FamilyDB" });

await this.client.CreateDocumentCollectionIfNotExistsAsync
(UriFactory.CreateDatabaseUri("FamilyDB"),
new DocumentCollection { Id = "FamilyCollection" });
```

Querying DocumentDB API

Creating a document

```
private async Task CreateFamilyDocumentIfNotExists(string databaseName,
string collectionName, Family family) { try { await
this.client.ReadDocumentAsync(UriFactory.CreateDocumentUri(databaseName,
collectionName, family.Id));

this.WriteToConsoleAndPromptToContinue("Found {0}", family.Id); } catch
(DocumentClientException de) { if (de.StatusCode ==
HttpStatusCode.NotFound) { await
this.client.CreateDocumentAsync(UriFactory.CreateDocumentCollectionUri(data
baseName, collectionName), family);

this.WriteToConsoleAndPromptToContinue("Created Family {0}", family.Id); }
else { throw; } } }
```

Querying DocumentDB API

Creating a document

Since DocumentDB is dynamic you just throw data in

```
await
client.CreateDocumentAsync(documentCollection.SelfLink,
listing);
```


Querying DocumentDB API

Querying

- Everything is done **asynchronously** in the SDK
- The ID of a new database is the friendly name
- Everything references the “SelfLink”

  This is the internal ID of the resource you are working with

  Used to build up the API call

Querying DocumentDB API

Querying: Simple

- SELECT * FROM

```
var client = GetClient();
var collection = await GetCollection(client, Keys.ListingsDbName,
Keys.ListingDbCollectionName);

string sql = String.Format("SELECT * FROM {0}",
Keys.ListingDbCollectionName);

var jeepsQuery =
client.CreateDocumentQuery<Listing>(collection.SelfLink,
sql).ToArray();
var jeeps = jeepsQuery.ToArray();
```

Querying DocumentDB API

Querying: More Complex

```
var client = GetClient();
var collection = await GetCollection(client, Keys.ListingsDbName,
Keys.ListingDbCollectionName);

string sql = String.Format(@"SELECT l.Color, l.Options, l.Package,
l.Type, l.Image,
l.Dealer, l.Id
FROM {0} l
JOIN o IN l.Options
WHERE o.Name = 'hard top'", Keys.ListingDbCollectionName);
var hardtopQuery =
client.CreateDocumentQuery<Listing>(collection.SelfLink,
sql).ToArray();
```

Querying DocumentDB API

Querying: JOINS

- DocumentDB deals with the denormalized data model of schema-free documents
- This is the logical equivalent of a "self-join".
- You can have up to 2 joins. Ask Microsoft support if you need more than 2 joins

Querying DocumentDB API

Querying: JOINS

```
SELECT
    f.id AS familyName,
    c.givenName AS childGivenName,
    c.firstName AS childFirstName,
    p.givenName AS petName
FROM Families f
JOIN c IN f.children
JOIN p IN c.pets
```


Querying DocumentDB API

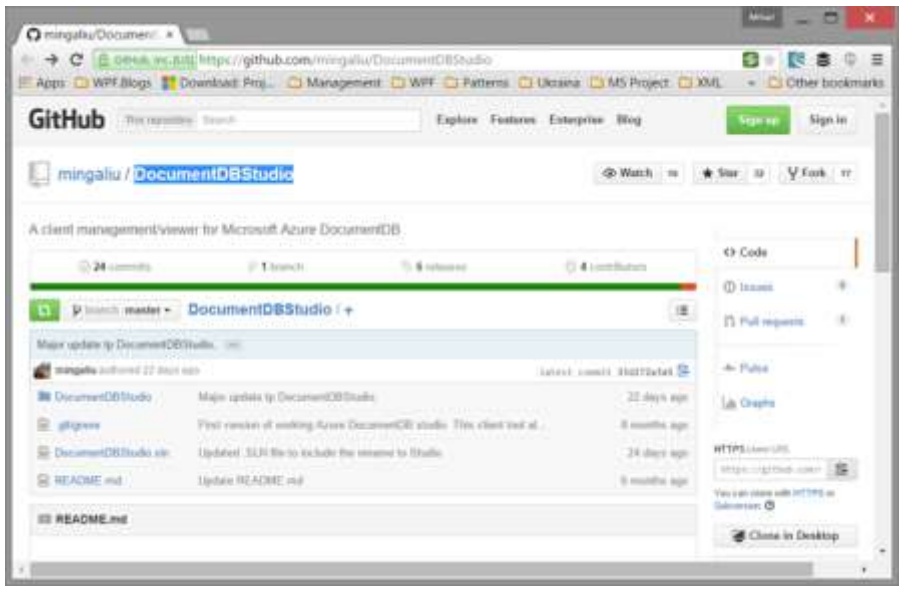
Querying: DEMO

- Live DEMO – Document Explorer
  http://www.documentdb.com/sql/demo

Querying DocumentDB API

UDF : User Defined Functions

- DocumentDB Studio
  https://github.com/mingaliu/DocumentDBStudio



Querying DocumentDB API

UDF : User Defined Functions

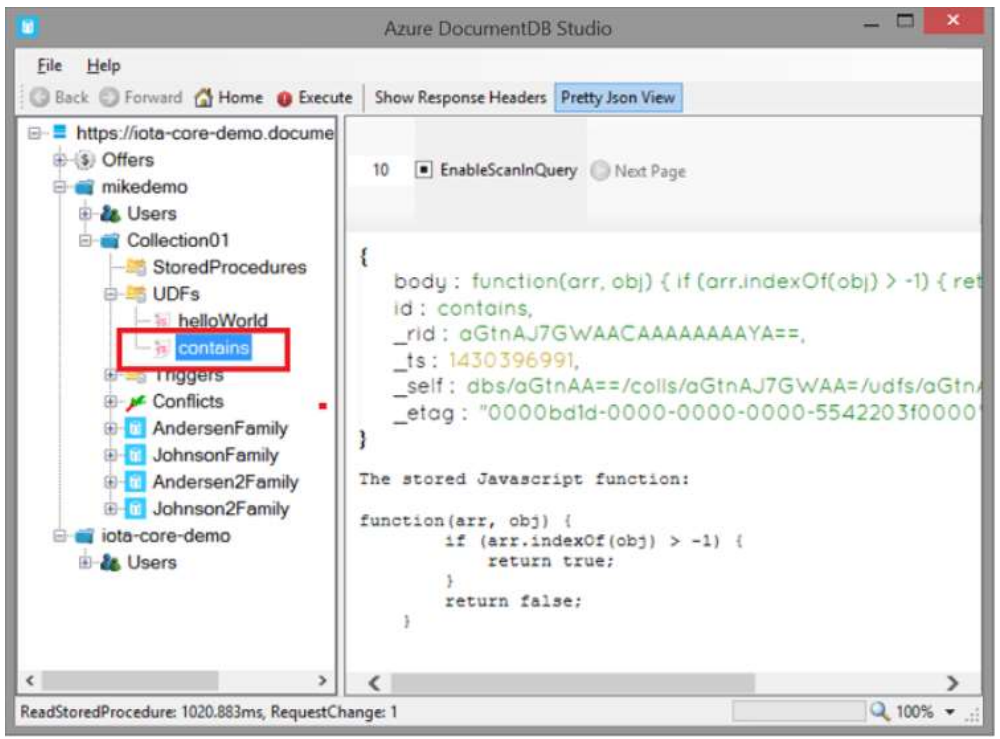
- Create an UDF

```
var containsUdf = {
    id: "contains",
    body: function(arr, obj) {
        if (arr.indexOf(obj) > -1) {
            return true;
        }
        return false;
    }
};
```


Querying DocumentDB API

UDF : User Defined Functions

- Create an UDF



Querying DocumentDB API

UDF : User Defined Functions

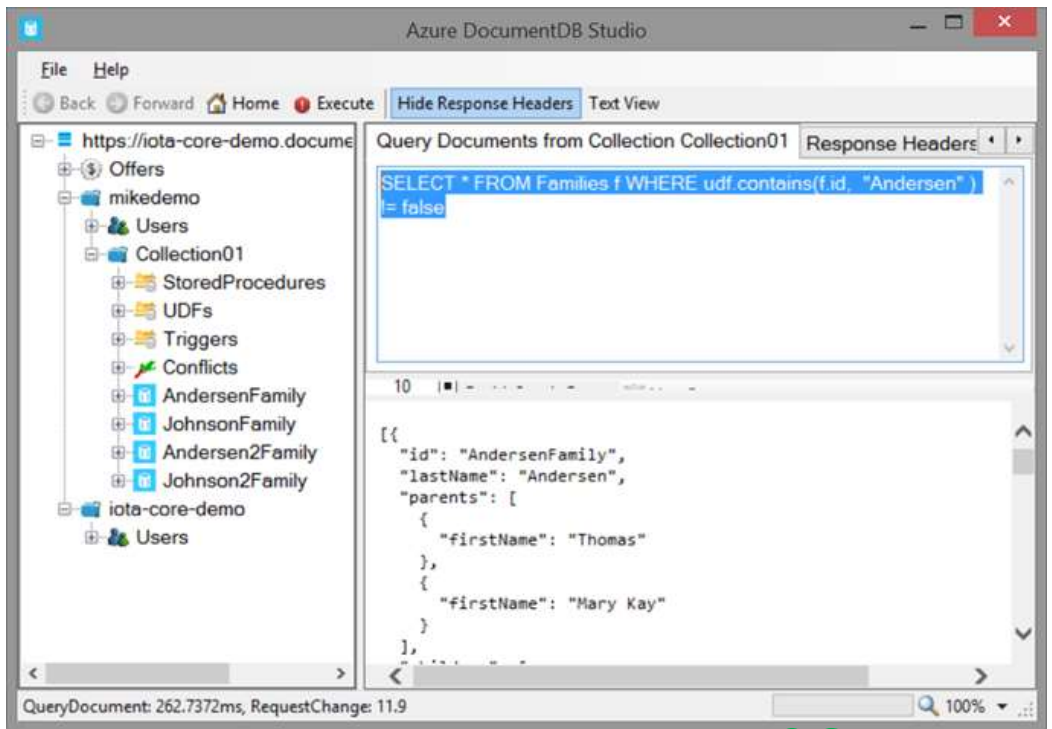
- Using UDF in Queries

```
SELECT * FROM Families f WHERE
udf.contains(f.id,  "Andersen" ) != false
```


Querying DocumentDB API

UDF : User Defined Functions

- Result after execution of queries using UDF functions



Querying DocumentDB API

Stored Procedures:

- Stored Procedures should be written in
- It is not possible to call a Stored Procedure from SQL Statement
- How to create a Stored Procedure

Querying DocumentDB API

Stored Procedures: Create a Stored Procedure

```
var createDocumentStoredProc = {
    id: "createCustomDocument",
    body: function createCustomDocument(documentToCreate) {
        var context = getContext();
        var collection = context.getCollection();

        var accepted = collection.createDocument(collection.getSelfLink(),
            documentToCreate,
            function (err, documentCreated) {
                if (err) throw new Error('Error' + err.message);
                context.getResponse().setBody(documentCreated.id)
            });
        if (!accepted) return;
    }
}
```


Querying DocumentDB API

Stored Procedures: Execute a Stored Procedure

```
var result =
client.ExecuteStoredProcedureAsync(
createdStoredProcedure._self);
```


Querying DocumentDB API

Triggers

- Pre-Triggers
- Post-Triggers

Querying DocumentDB API

Pre-Triggers

- DocumentDB provides triggers that are executed or triggered by an operation on a document.
- Pre-triggers cannot have any input parameters.

Querying DocumentDB API

Pre-Triggers: Create

```
var validateDocumentContentsTrigger = {
    name: "validateDocumentContents",
    body: function validate() {
        var context = getContext();
        var request = context.getRequest();
        // document to be created in the current operation
        var documentToCreate = request.getBody();
        // validate properties
        if (!("timestamp" in documentToCreate)) {
            var ts = new Date();
            documentToCreate["doc timestamp"] = ts.getTime();
        }
        // update the document that will be created
        request.setBody(documentToCreate);
    },
    triggerType: TriggerType.Pre,
    triggerOperation: TriggerOperation.Create
}
```


Querying DocumentDB API

Pre-Triggers: Using

{**id** : Pre-trigger-test-document,
doc-timestamp : 1430497343363,
_rid : SEZmALv1TAAKAAAAAAAAAA==,
_ts : 1430497343,
_self :
dbs/SEZmAA==/colls/SEZmALv1TAA=/docs/SEZmALv1TAAKAAAAAAAAAA==/,
_etag : "00001b00-0000-0000-0000-5543a83f0000",
_attachments : attachments/
}

Querying DocumentDB API

Post-Triggers:

- Post-triggers, like pre-triggers, are associated with an operation on a document and don't take any input parameters.
- They run after the operation has completed, and have access to the response message that is sent to the client.

Querying DocumentDB API

Post-Triggers:

- **Transactional execution of triggers** in CosmosDB
- This post-trigger runs as part of the same transaction as the creation of the original document
- If there is an exception from the post-trigger, the whole transaction will fail and be rolled back

Querying DocumentDB API

Post-Triggers:

```
var updateMetadataTrigger = {
    name: "updateMetadata",
    body: function updateMetadata() {
        var context = getContext();
        var collection = context.getCollection();
        var response = context.getResponse();
        var createdDocument = response.getBody();
        var filterQuery = 'SELECT * FROM c WHERE c.id = "_metadata"';
        var accept = collection.queryDocuments(collection.getSelfLink(),
filterQuery,
            updateMetadataCallback);
        if(!accept) throw "Unable to update metadata, abort";
        ....
    },
    triggerType: TriggerType.Post,
    triggerOperation: TriggerOperation.All
}
```


Querying DocumentDB API

Post-Triggers:

- This trigger queries for the metadata document and updates it with details about the newly created document.

Querying DocumentDB API

Post-Triggers:

```
function updateMetadataCallback(err, documents, responseOptions) {
    var metadataDocument = documents[0];
        // update metadata
        metadataDocument.createdDocuments += 1;
        metadataDocument.createdNames += " " + createdDocument.id;
        var accept = collection.replaceDocument(metadataDocument._self,
            metadataDocument, function(err, docReplaced) {
                if(err) throw "Unable to update metadata, abort";
            });
        if(!accept) throw "Unable to update metadata, abort";
        return;
    }
```


Querying DocumentDB API

Transactions

- Transaction in a typical database can be defined as a sequence of operations performed as a single logical unit of work.
- Each transaction provides ACID guarantees.
  ACID is a well-known acronym that stands for four properties - Atomicity, Consistency, Isolation and Durability.

Querying DocumentDB API

Transactions: Commit and rollback

- Transactions are integrated into DocumentDB's JavaScript programming model.
- Inside a JavaScript function, all operations are automatically wrapped under a single transaction

Querying DocumentDB API

Transactions: Commit and rollback

- If the JavaScript completes without any exception, the operations to the database are committed
- “BEGIN TRANSACTION” and “COMMIT TRANSACTION” statements in relational databases are implicit in DocumentDB

CosmosDB Performance

CosmosDB Performance: DocumentDB API

Networking Connection Policy : Modes

- Gateway Mode (default)
- Direct Mode

Use direct connection mode for better performance

Direct Mode is currently supported only in the .NET SDK, but will be available in other platforms with subsequent SDK refreshes

CosmosDB Performance: DocumentDB API

Networking Connection Policy: Protocol

- TCP
- HTTPS

Both protocols are RESTful

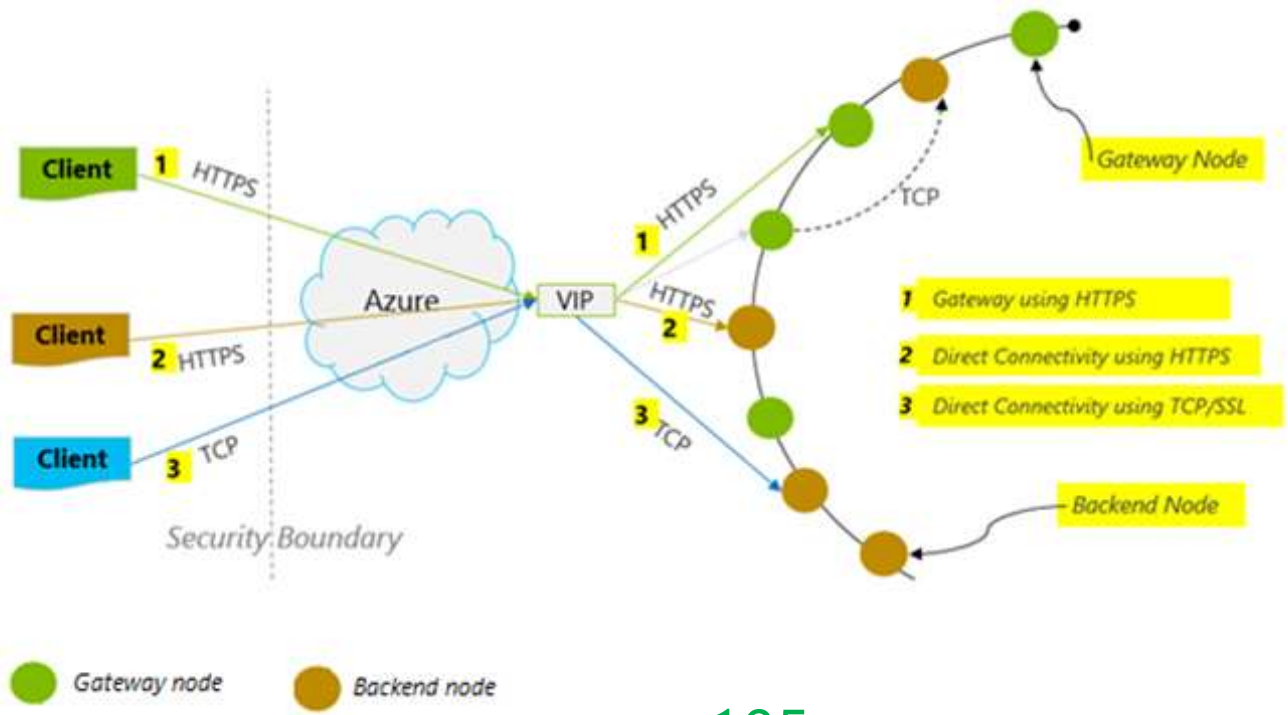
Use TCP protocol for better performance

```
var client =DocumentClient client = new DocumentClient
(serviceEndpoint, authKey,
        new ConnectionPolicy
        {
            ConnectionMode = ConnectionMode.Direct,
            ConnectionProtocol = Protocol.Tcp
        });
```

CosmosDB Performance: DocumentDB API

Networking Connection Policy:

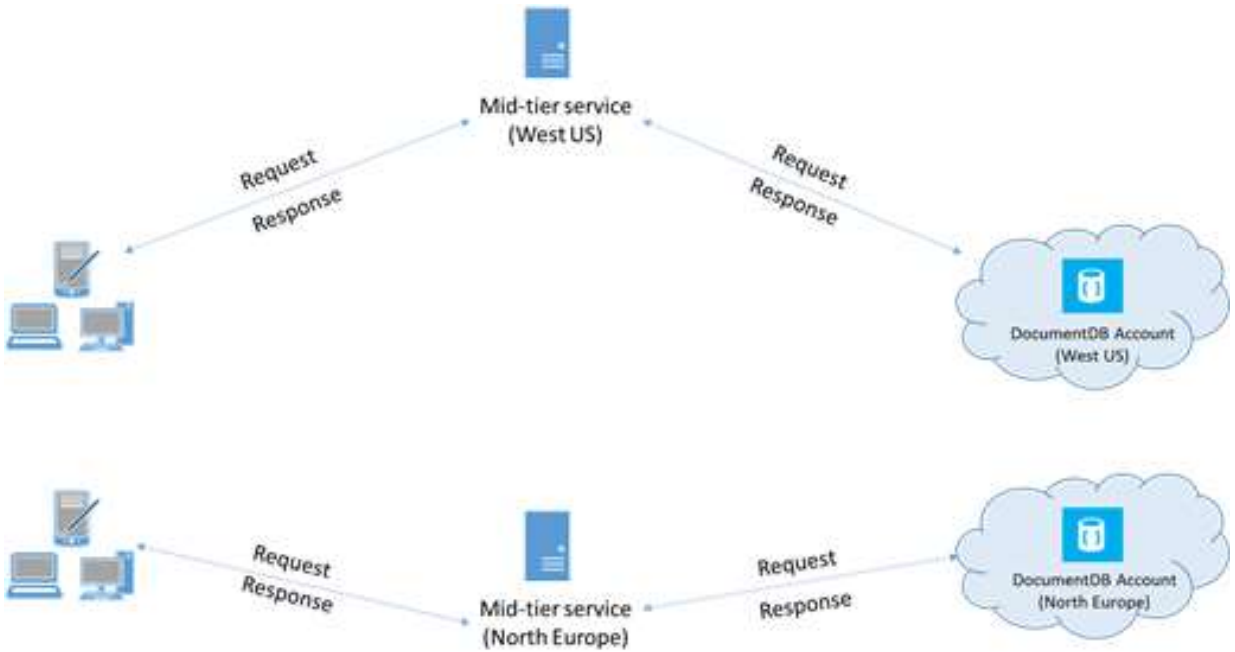
Direct Mode is currently supported only in the .NET SDK, but will be available in other platforms with subsequent SDK refreshes.



CosmosDB Performance: DocumentDB API

Networking Tip :

- Collocate clients in same Azure region for performance



CosmosDB Performance: DocumentDB API

SDK Usage Tip :

- Cache document and collection SelfLinks for lower read latency

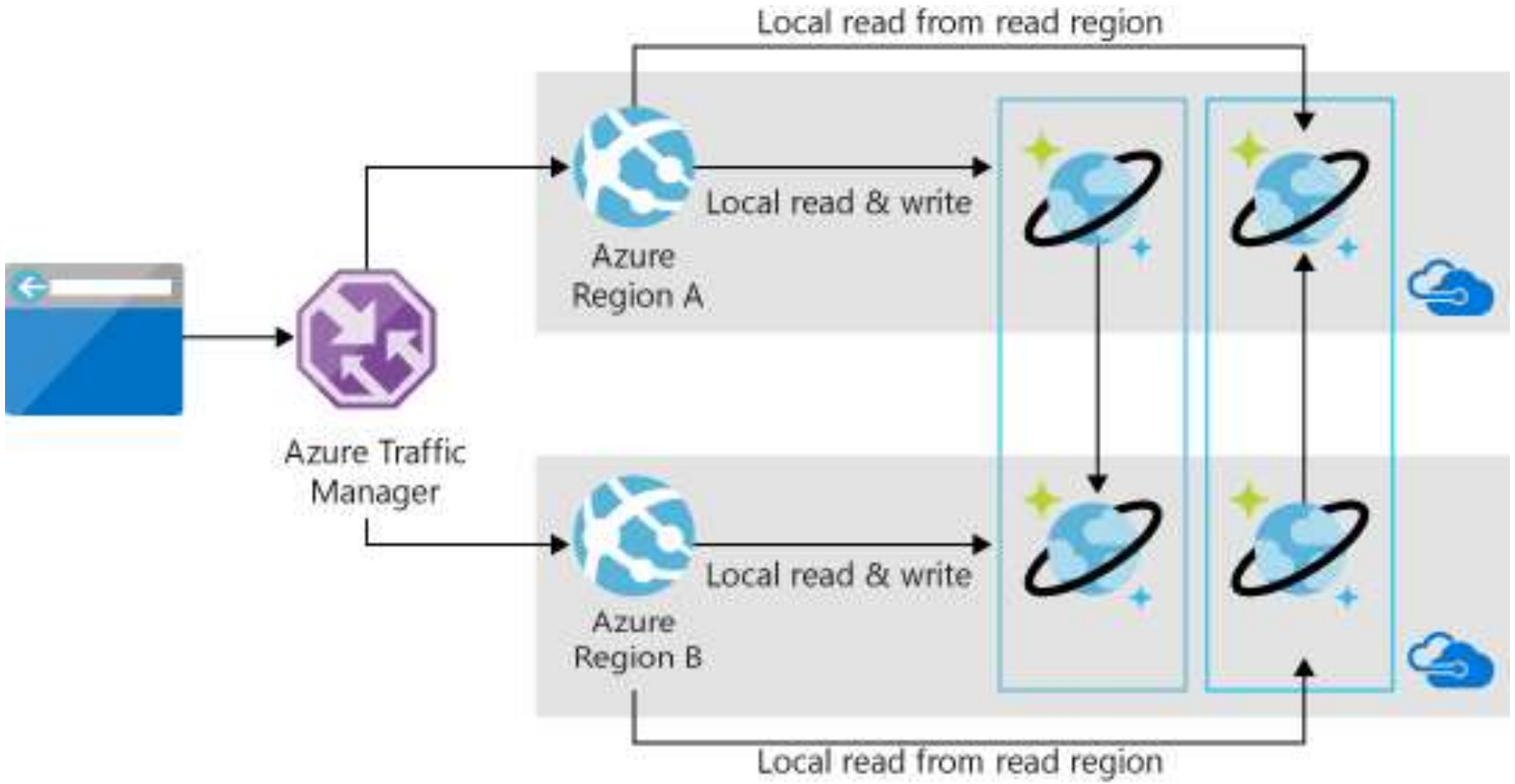
```
Document document = await
client.ReadDocumentAsync("/dbs/1234/colls/1234354/docs/2332435465");
```

CosmosDB Regional Failover

Cosmos DB supports both explicit and policy driven failovers:

- How do manual failovers work in Cosmos DB?
- How do automatic failovers work in Cosmos DB and what happens when a data center goes down?
- How can you use manual failovers in application architectures?

CosmosDB Multi-master globally replicated DB



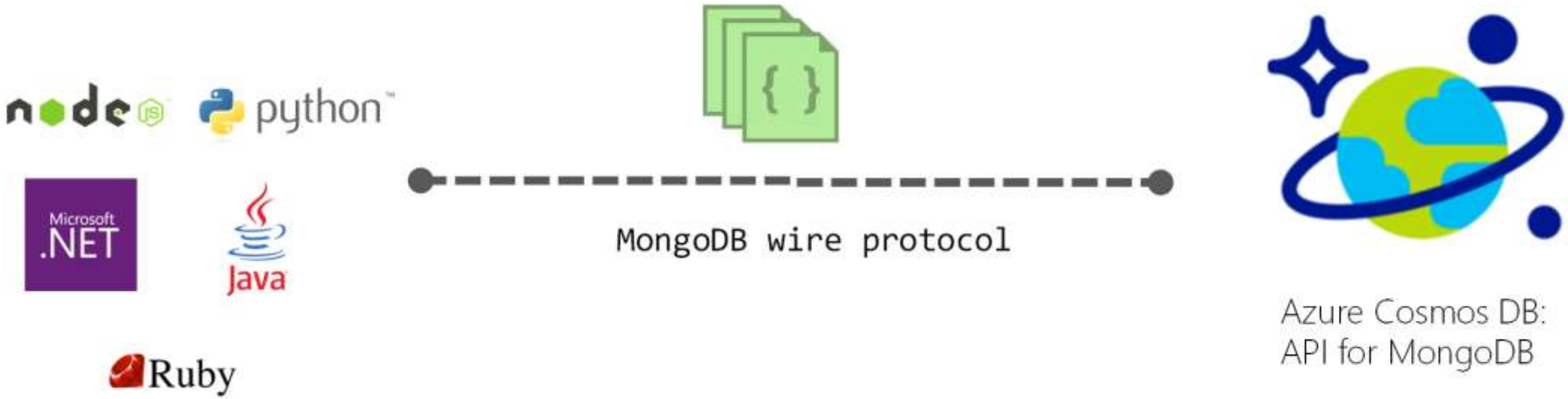
CosmosDB and MongoDB

Cosmos DB: API for MongoDB

- Cosmos DB databases can be used as the data store for apps written for MongoDB
- By using existing drivers, your application written for MongoDB can now communicate with Cosmos DB and use Cosmos DB databases instead of MongoDB databases.

Cosmos DB: API for MongoDB

- You can easily build and run MongoDB database applications in the Azure cloud with Azure Cosmos DB



Cosmos DB: API for MongoDB

The benefit of using Azure Cosmos DB for MongoDB:

- Elastically scalable throughput and storage:
- Multi-region replication:
- MongoDB compatibility
- No server management:
- Tunable consistency levels
- Automatic indexing
- Enterprise grade

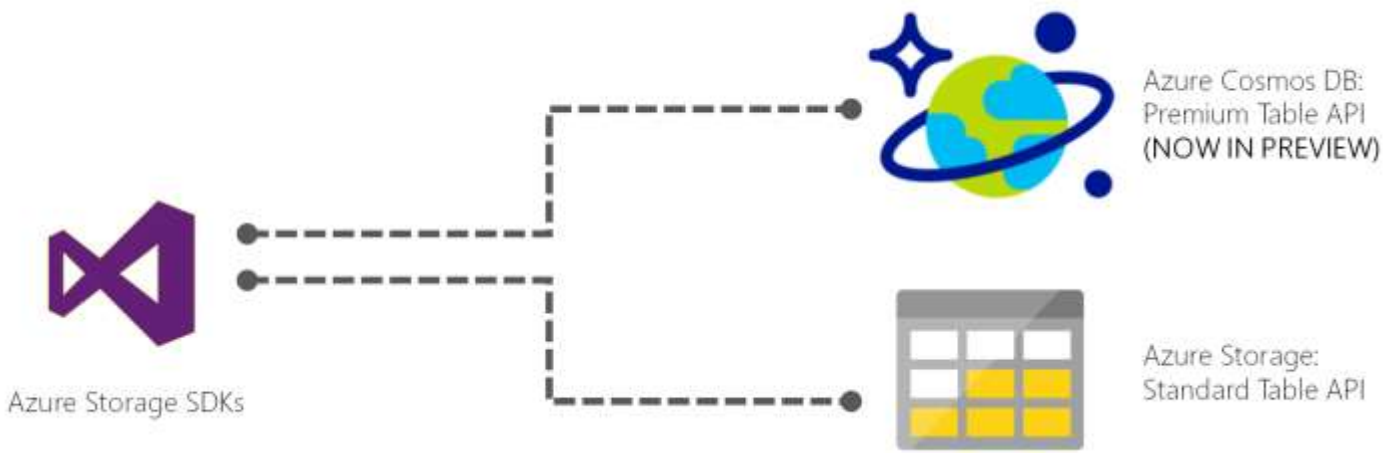
CosmosDB and Table Storage

Cosmos DB: Table API

- Content Azure Cosmos DB provides the Table API (preview) for applications that need a key-value store with flexible schema, predictable performance, global distribution, and high throughput
- The Table API provides the same functionality as Azure Table storage, but leverages the benefits of the Azure Cosmos DB engine.

Cosmos DB: Table API

- DB will introduce support for storage-optimized tables in a future update, and existing and new Azure Table storage accounts will be upgraded to Azure Cosmos DB.



Cosmos DB: Table API

- Azure Cosmos DB provides the Table API (preview): for applications written for the Azure Table storage service and need premium capabilities
  - turn-key global distribution
  - dedicated throughput worldwide
  - single-digit millisecond latencies at the 99th percentile
  - guaranteed high availability
  - automatic secondary indexing.

These applications can migrate to Azure Cosmos DB using the Table API with no code changes, and take advantage of premium capabilities.

Cosmos DB: Table API

- Premium and standard Table APIs

	Azure Table Storage	Azure Cosmos DB: Table storage (preview)
Latency	Fast, but no upper bounds on latency	Single-digit millisecond latency for reads and writes, backed with <10-ms latency reads and <15-ms latency writes at the 99th percentile, at any scale, anywhere in the world
Throughput	variable throughput model. Tables have a scalability limit of 20,000 operations/s	Highly scalable with dedicated reserved throughput per table, that is backed by SLAs. Accounts have no upper limit on throughput, and support >10 million operations/s per table
Global Distribution	Single region with one optional readable secondary read region for HA. You cannot initiate failover	Turn-key global distribution from one to 30+ regions, Support for automatic and manual failovers at any time, anywhere in the world
Indexing	Only primary index on PartitionKey and RowKey. No secondary indexes	Automatic and complete indexing on all properties, no index management

Cosmos DB: Table API

- Premium and standard Table APIs

	Azure Table Storage	Azure Cosmos DB: Table storage (preview)
Query	Query execution uses index for primary key, and scans otherwise.	Queries can take advantage of automatic indexing on properties for fast query times. Azure Cosmos DB's database engine is capable of supporting aggregates, geo-spatial, and sorting.
Consistency	Strong within primary region, Eventual with secondary region	Five well-defined consistency levels to trade off availability, latency, throughput, and consistency based on your application needs
Pricing	Storage-optimized	Throughput-optimized
SLAs	99.9% availability	99.99% availability within a single region, and ability to add more regions for higher availability. Industry-leading comprehensive SLAs on general availability

Cosmos DB: Table API

- Quering Table API.
  - Query on PartitionKey and RowKey

https://<mytableendpoint>/People
(PartitionKey='Harp',RowKey='Walter')

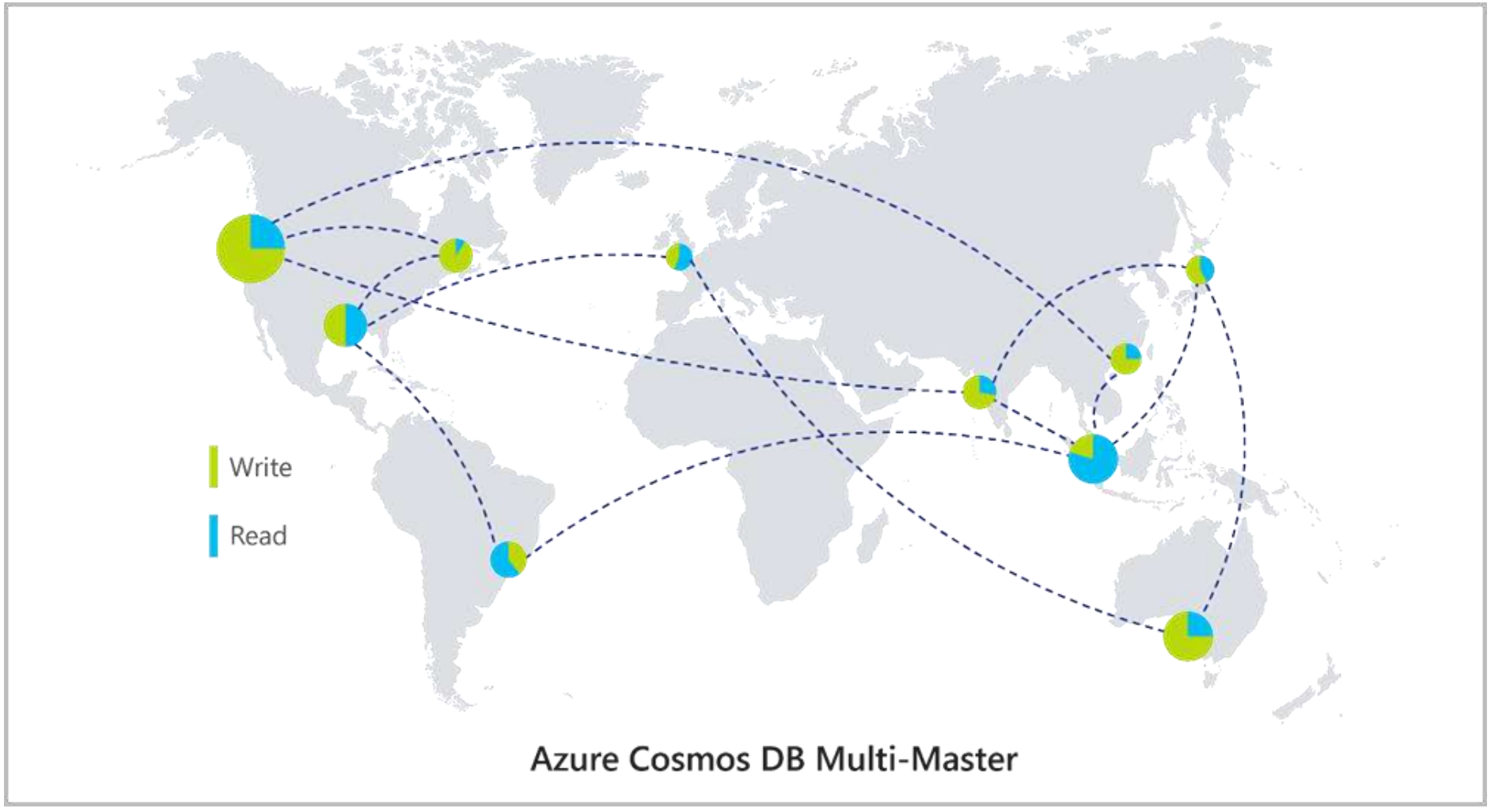
  - Query by using an OData filter

https://<mytableapi-endpoint>/People()?$filter=
PartitionKey%20eq%20'Smith'%20and%20Email%20eq%
20'Ben@contoso.com'

Multi-master at global scale with Azure Cosmos DB

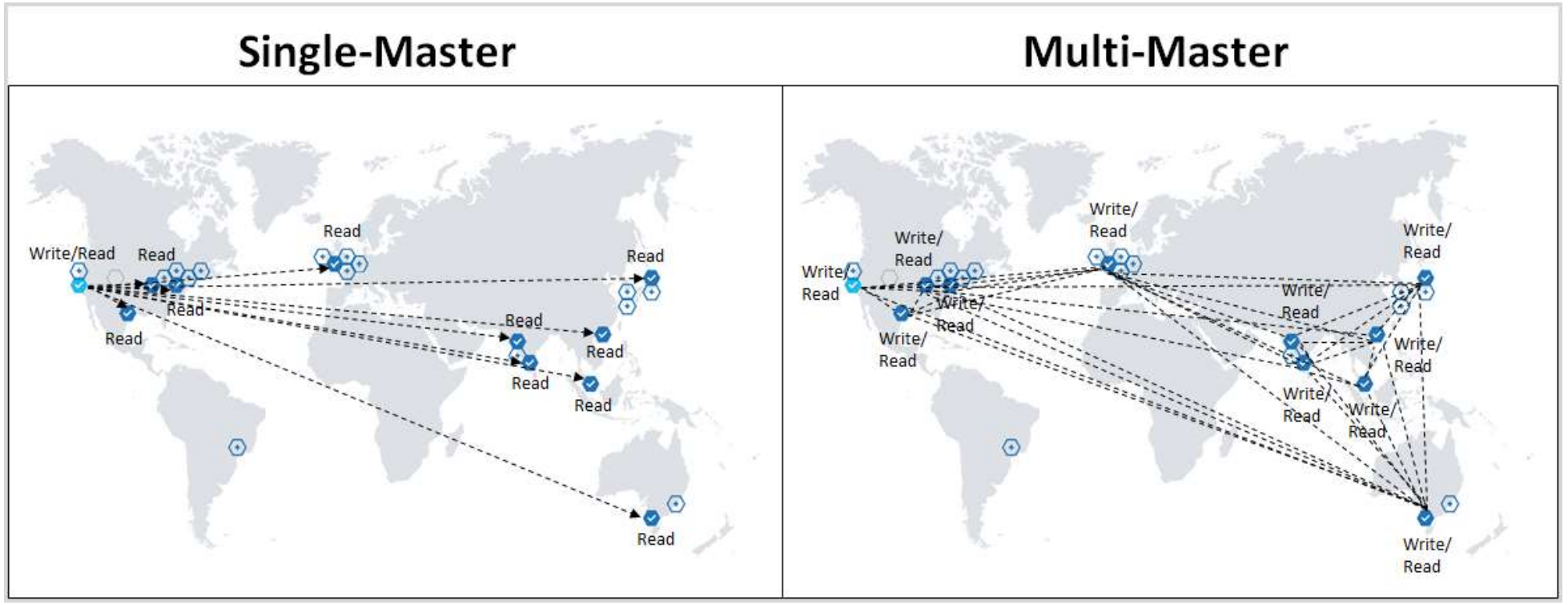
- You can update data in any region that is associated with your database account
- Data updates can propagate asynchronously
- Azure Cosmos DB you get write latency of <10 ms at the 99th percentile anywhere in the world, 99.999% write and read availability anywhere in the world

Multi-master at global scale with Azure Cosmos DB



Multi-master at global scale with Azure Cosmos DB

- Benefits of having multi-master support



CosmosDB and Graph Storage

Cosmos DB: Table API

- Quering Table API.
  - Query by using LINQ

```
CloudTableClient tableClient = account.CreateCloudTableClient();
CloudTable table = tableClient.GetTableReference("people");
TableQuery<CustomerEntity> query = new TableQuery<CustomerEntity>()
    .Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(PartitionKey, QueryComparisons.Equal,
"Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(Email,
QueryComparisons.Equal,"Ben@contoso.com")
    ));
await table.ExecuteQuerySegmentedAsync<CustomerEntity>(query, null);
```

Cosmos DB: Graph API

- Graph modeling
- Traversal APIs
- Turn-key global distribution
- Elastic scaling of storage and throughput with less than 10 ms read latencies and less than 15 ms at the 99th percentile
- Automatic indexing with instant query availability
- Tunable consistency levels
- Comprehensive SLAs including 99.99% availability

Cosmos DB: Graph API

To query Azure Cosmos DB, you can use

- Apache TinkerPop graph traversal language
- Gremlin
- other TinkerPop-compatible graph systems like Apache Spark GraphX.

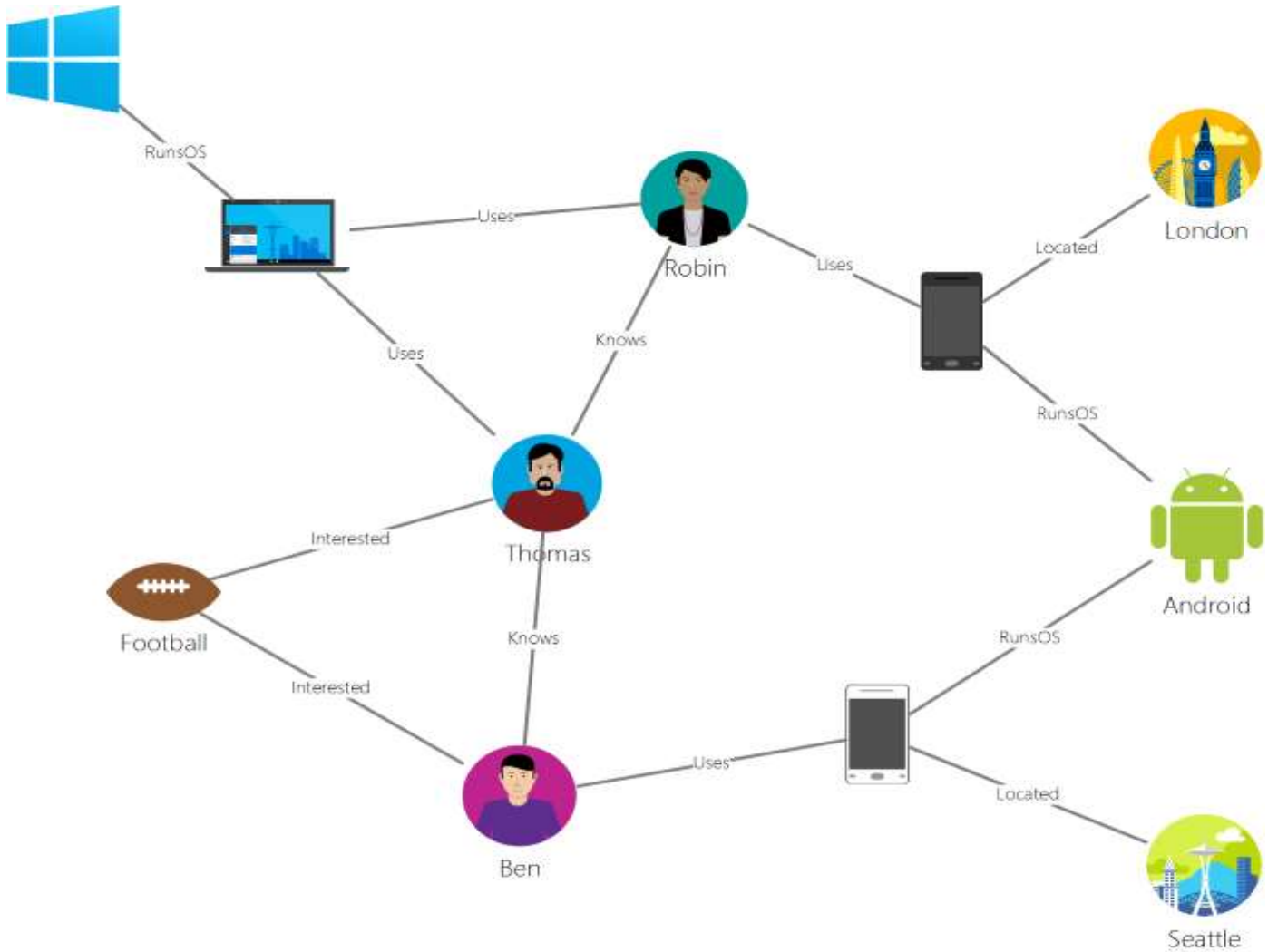
Cosmos DB: Graph API

Graph database

- A graph is a structure that's composed of vertices and edges. Both vertices and edges can have an arbitrary number of properties.
- Properties express information about the vertices and edges.
- Example properties include a vertex that has a name, age, and edge, which has a timestamp and/or a weight.
- This model is known as a property graph.
- Azure Cosmos DB supports the property graph model.

Cosmos DB: Graph API

Graph database



Cosmos DB: Graph API

- Quering Graph API.
  - Create a document Clinent

```
using (DocumentClient client = new DocumentClient(
    new Uri(endpoint),
    authKey,
    new ConnectionPolicy { ConnectionMode =
ConnectionMode.Direct, ConnectionProtocol =
Protocol.Tcp }))
```

Cosmos DB: Graph API

- Quering Graph API.
  - Create a database

```
Database database = await
client.CreateDatabaseIfNotExistsAsync
(new Database { Id = "graphdb" });
```


Cosmos DB: Graph API

- Quering Graph API.
  - Create a graph

```
DocumentCollection graph = await
client.CreateDocumentCollectionIfNotExistsAsync(
    UriFactory.CreateDatabaseUri("graphdb"),
    new DocumentCollection { Id = "graph" },
    new RequestOptions { OfferThroughput = 1000 });
```

Cosmos DB: Graph API

- Quering Graph API.
  - Create a query

```
// The CreateGremlinQuery method extensions
//allow you to execute Gremlin queries and iterate
// results asychronously
IDocumentQuery<dynamic> query =
client.CreateGremlinQuery<dynamic>(graph, "g.V().count()");
while (query.HasMoreResults)
{
    foreach (dynamic result in await query.ExecuteNextAsync())
    {
        Console.WriteLine($"\t {JsonConvert.SerializeObject(result)}");
    }
}
```

Demos

CosmosDB Demos

Dealing with CosmosDB

Raffle and goodbye Beer

Remember to **visit the sponsors**, stay for **the raffle** and **goodbye beers** ☺

Join our sponsors for a lunch break session in :
cust 0.01 and **cust 1.06**

We hope you'll all have a great Saturday.

Regis, Kenneth

BIG Thanks to SQL Sat Denmark sponsors

GOLD

kapacity Business Analytics

TIMEXTENDER

Quest

Microsoft

SILVER

NEXTAGENDA
BUSINESS INTELLIGENCE LØSNINGER

BRONZE

ORANGEMAN

accobat
Know your business