**Agenda: Azure Cosmos DB Service**

* What is Cosmos DB
* Createing Cosmos Db Account using Azure Portal.
* Auto Indexing
* Managing Throughput using RU's
* Global Replication and Consistency Levels
* Horizontal Scaling using Patitioning
* Understanding SQL Model database Schema
* Creating and Saving Documents using Portal
* Programming Schema
* Adding / Editing / Deleting and Reading

**What is Cosmos DB?**

* What is NoSQL Database
  + **Cosmos DB**
  + Mongo DB
  + Cassandra
  + Apache HBase
  + Amazon Dynamo DB
  + Orient DB
  + Arrango DB

**Todays Requirement:** 3Vs = Volume / Variety / Velocity



**NoSQL Database are**

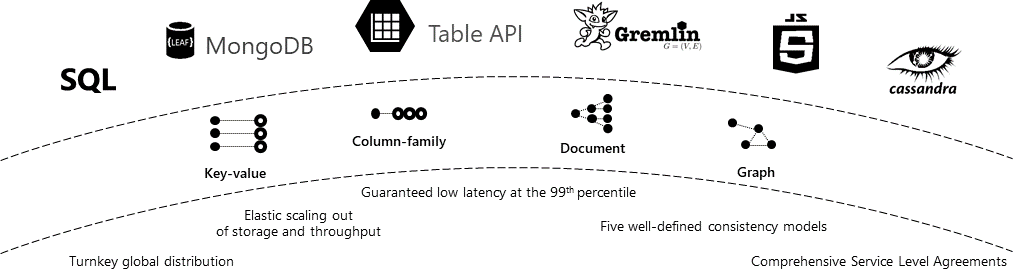
1. **Distributed** = Replicas ensure High Throughput / Availability and Low Latency.
2. **Scale-out** = Horizontal Partitioning enables virtually limitless storage and throughput.
3. **Schema free** = Document, table, graph and columnar Data Model.

**What is Cosmos DB**

* It's an evolution of Document DB which was publicly available from 2015. Document DB is now referred as SQL API and is just one of the API's supported in Cosmos DB.
* Introduced in the year 2017.
* Azure Cosmos DB is **Microsoft's globally distributed, multi-model/multi-api database**.
* Azure Cosmos DB enables you to elastically and independently **scale throughput and storage** across any number of Azure's geographic regions.
* **It offers throughput, latency, availability, and consistency** **guarantees** with comprehensive [service level agreements](https://aka.ms/acdbsla) (SLAs), something no other database service can offer. 99.99% availability SLA for all single region database accounts, and all 99.999% read availability on all multi-region database accounts.
* For a typical 1KB item, Cosmos DB guarantees end-to-end latency of reads under 10 ms and indexed writes under 15 ms at the 99th percentile, within the same Azure region. The median latencies are significantly lower (under 5 ms).
* **Five to ten times** more **cost effective** than a non-managed solution or an on-prem NoSQL solution. Three times cheaper than AWS DynamoDB or Google Spanner.

**Multiple data models and popular APIs for accessing and querying data**

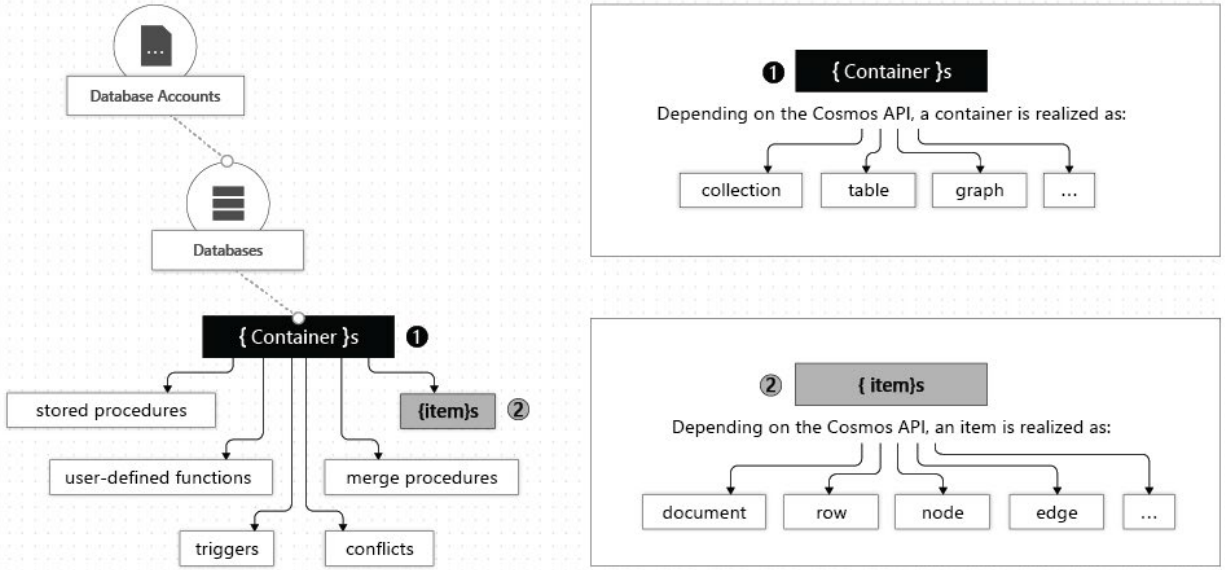
* The **atom-record-sequence** (ARS) based data model that Azure Cosmos DB is built on natively supports multiple data models, including but not limited to document, graph, key-value, table, and column-family data models.



* APIs for the following data models are supported with SDKs available in multiple languages:
  1. **SQL API (Core API)**:
  2. JavaScript and JavaScript Object Notation (JSON) native API based on the Azure Cosmos DB database engine.
  3. Provides query capabilities rooted in SQL
  4. Queries for documents based on their identifiers or make deeper queries based on properties of the document, complex objects, or the existence of specific properties
  5. Supports the execution of JavaScript logic within the database in the form of stored procedures, triggers, and user-defined functions.
  6. **MongoDB API**: It's based on BSON Document format. A massively scalable MongoDB-as-a-Service powered by Azure Cosmos DB platform. Compatible with existing MongoDB libraries, drivers, tools, and applications.
  7. **Table API**: A key-value database service built to provide premium capabilities (for example, automatic indexing, guaranteed low latency, global distribution) to existing Azure Table storage applications without making any app changes.
  8. **Gremlin API**: A fully managed, horizontally scalable **graph database** service that makes it easy to build and run applications that work with highly connected datasets supporting Open Graph APIs (based on the Apache TinkerPop specification, Apache Gremlin). It stores entities which are called **Nodes and Edges**. Recommended for new workloads that need to store relationships between data.
  9. **Cassandra API**: Globally distributed, Cassandra-as-a-Service powered by Azure Cosmos DB platform. Compatible with existing [Apache Cassandra](https://cassandra.apache.org/) libraries, drivers, tools, and applications.

## 

You can manage data in your account by creating databases, containers, and items. The following image shows the hierarchy of different entities in an Azure Cosmos DB account:



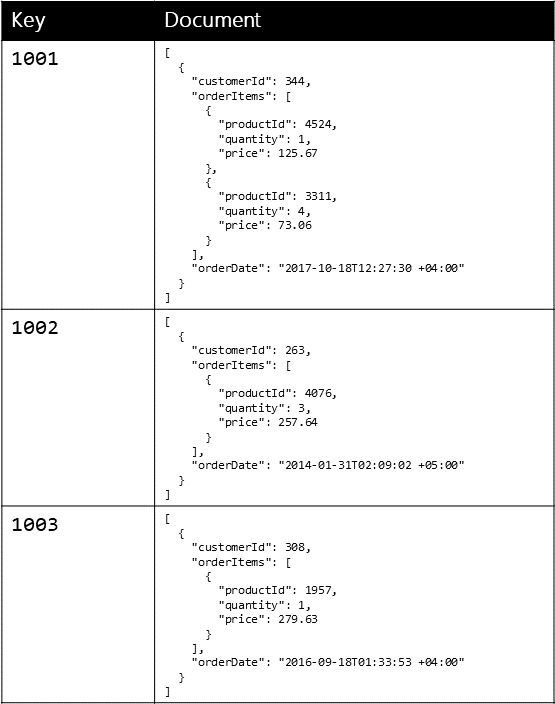
An Azure Cosmos **container** is specialized into API-specific entities as shown in the following table:

* **SQL Server = Table has Row**
* SQL API = Collection has Document
* Mongo DB = Collection has Document
* Table API = Table has Row
* Cassandra API = Table has Row
* Gremlin API = Graph has Node and Edge

***SQL DB database (Evolved version of DocumentDB)***

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. There are several ways in which you can encode the data in a document’s fields, including using Extensible Markup Language (XML), YAML, JavaScript Object Notation (JSON), Binary JSON (BSON), or even storing it as plain text. Unlike key/value stores, the fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

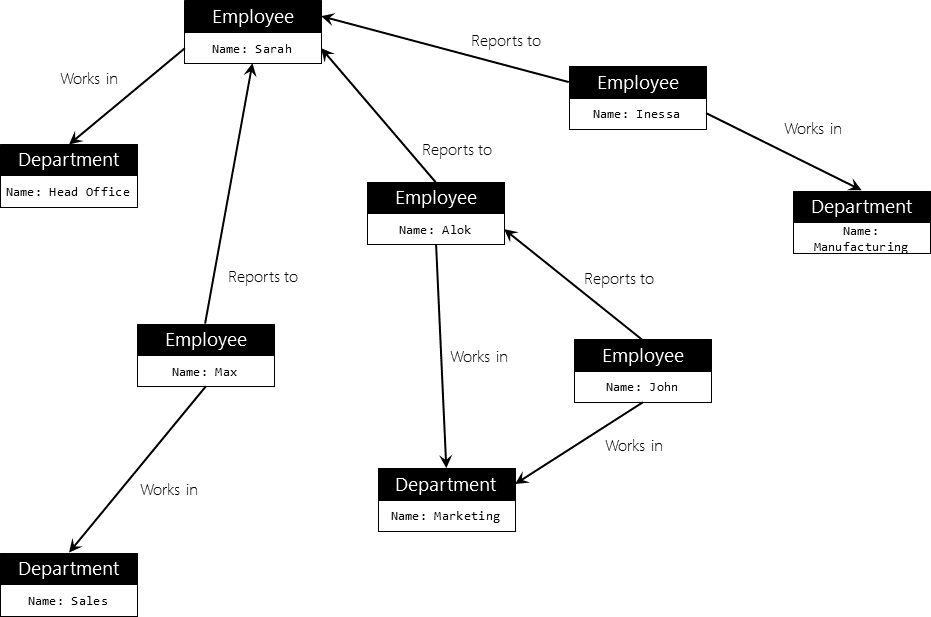
Typically, a document contains the entire data for an entity. What items constitute an entity are application specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an RDBMS.



#### **Graph databases**

A graph database stores two types of information, **nodes and edges**. You can think of nodes as entities. Edges which specify the relationships between nodes. Both nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

The purpose of a graph database is to allow an application to **efficiently perform queries** that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.



**Creating Cosmos DB Account using Portal**

1. Azure Portal 🡪 +Create a resource 🡪 Azure Cosmos DB
2. ID=dsscosmosdbacccount, API=**SQL**. . . 🡪 Create
3. **Add Collection**: Data Explorer 🡪 New Collection 🡪 Database Id=**Organization**, Collection Id=**Employees**, 🡪 OK
4. Add Sample data: Click on New Document

{

"empid": "1",

"name": "E1",

"department": "Development",

"isPermanent": false

}

{

"empid": "2",

"name": "E2",

"department": "Training",

"isPermanent": false

}

{

"empid": "3",

"name": "E1",

"department": "Development",

"isPermanent": true

}

1. We can as well update and delete documents from the same interface

Queries:

SELECT \* FROM c order by c.department desc

SELECT \* FROM d where c.department = 'development'

**Cosmos DB Local Emulator**

<http://aka.ms/cosmosdb-emulator>

and open the explorer in browser

**Azure Cosmos database migration toolkit:**

Using this tool, we can import data from JSON files, SQL, CSV files, MongoDB, Amazon DynamoDB, etc. You can download the migration tool source code from [this repository of the GITHUB](https://github.com/azure/azure-documentdb-datamigrationtool) and compile it locally, or you can download the pre-compiled library from [here](https://aka.ms/csdmtool)

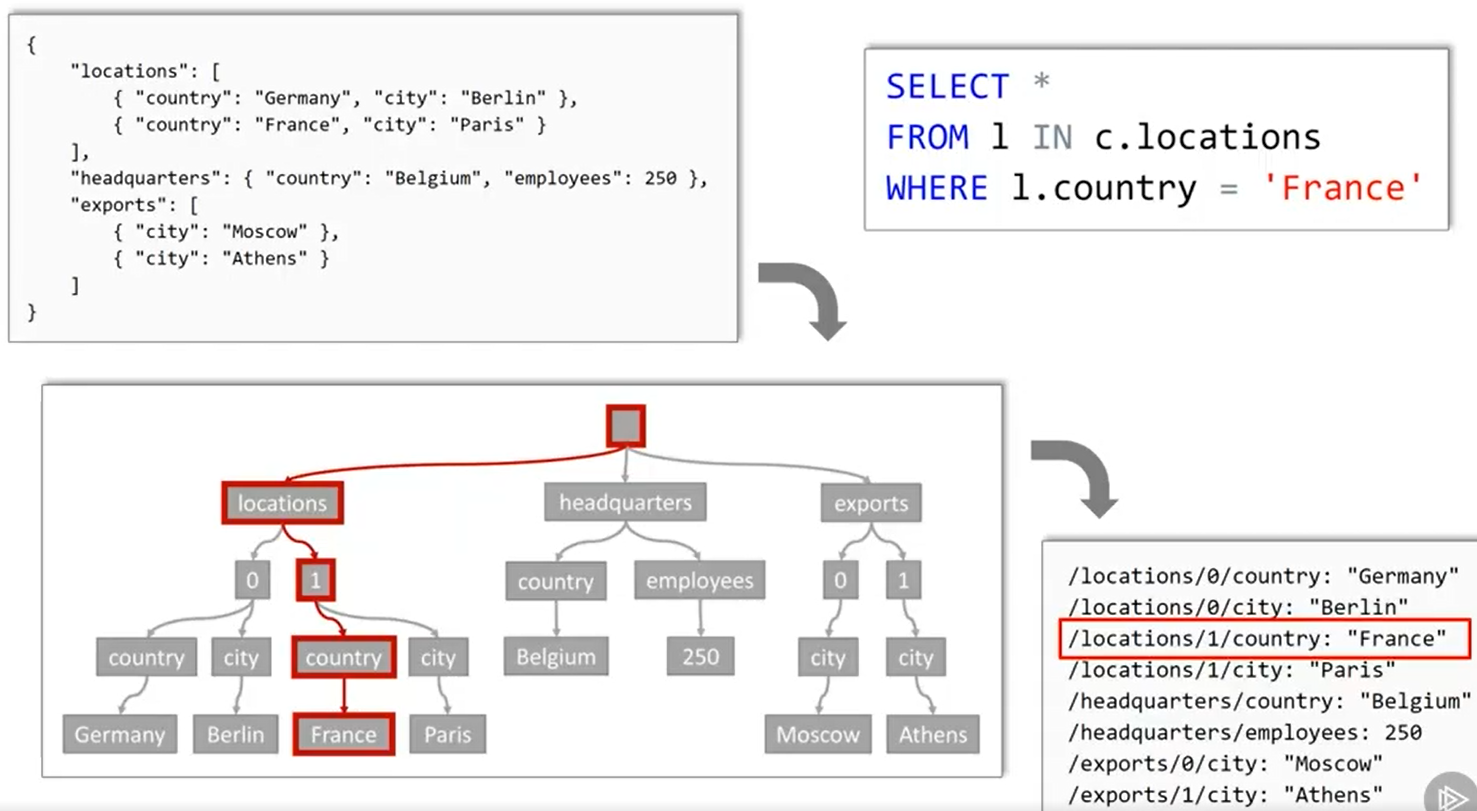
**Auto Indexing**

Cosmos Db by default index every property in every item.

**From the JSON object 🡪 Parses and creates a TREE 🡪 Creates flat key-values pairs and same are indexed.**

**Result:** Container with no defined schema but fully indexed.

This is called as **Inverted indexing** and overload is much lower than in SQL Schemas. This is supported in all Cosmos DB API.



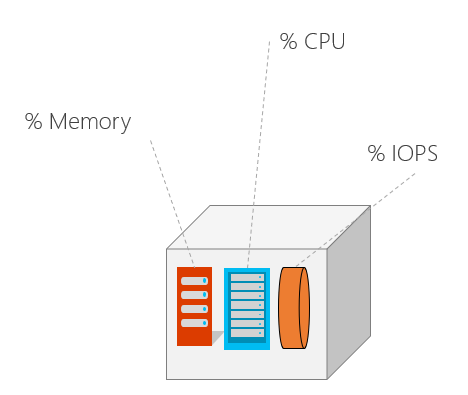
#### **Managing Throughput**

**Performance can be measured in two ways**

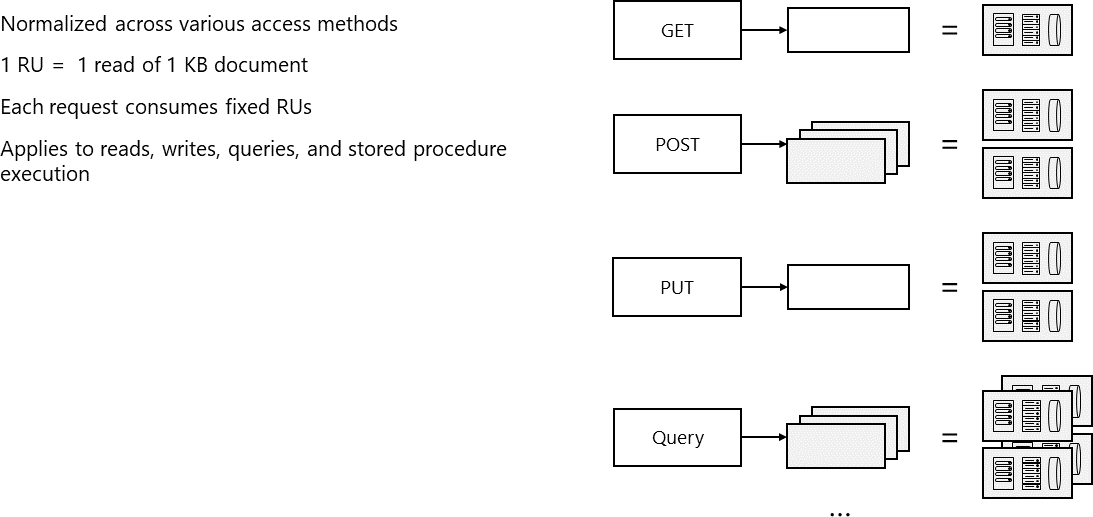
1. Latency means wait time – How fast is response for a given request.
2. Throughput means how many requests can be served within a specific period of time.

**About Request Unit (RU) in Cosmos DB**

* Azure Cosmos DB **reserves** resources to manage the throughput of an application. Because, application load and access patterns change over time, Azure Cosmos DB has support built-in to increase or decrease the amount of reserved throughput available at any time.
* With Azure Cosmos DB, reserved throughput is specified in terms of **request unit processing per second (RU/s)**. A request unit is a normalized measure of request processing cost.

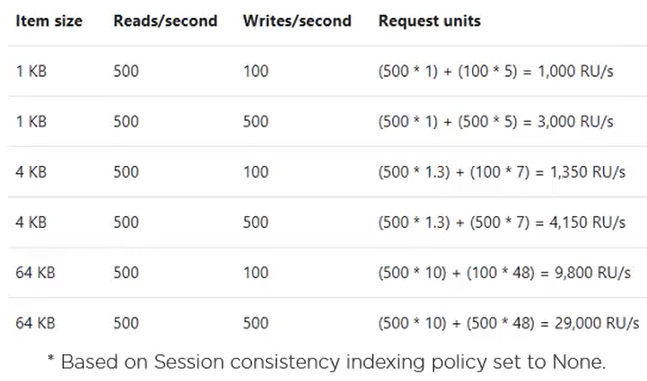


* You **reserve** several guaranteed request units to be available to your application **on a per-second basis**.
* Each operation in Azure Cosmos DB, including writing a document, performing a query, and updating a document, consumes CPU, memory, and Input/output operations per second (IOPS). That is, each operation incurs a request charge, which is expressed in request units.
* A single request unit represents the processing capacity that’s required to read, via **self-link** or ID, a single item that is 1 kilobyte (KB) and that consists of 10 unique property values (excluding system properties).
* A request to create (insert), replace, or delete the same item consumes **more processing** from the service and thereby requires more request units.
* Every CosmosDB response header shows the RU charge for that request.
* RU are deterministic, the same request will always require the same number of request units.



Note: Exceeding reserved throughput limit will result is request throttled or failure with status code 429 (success is 200).

**Estimating Throughput Needs**

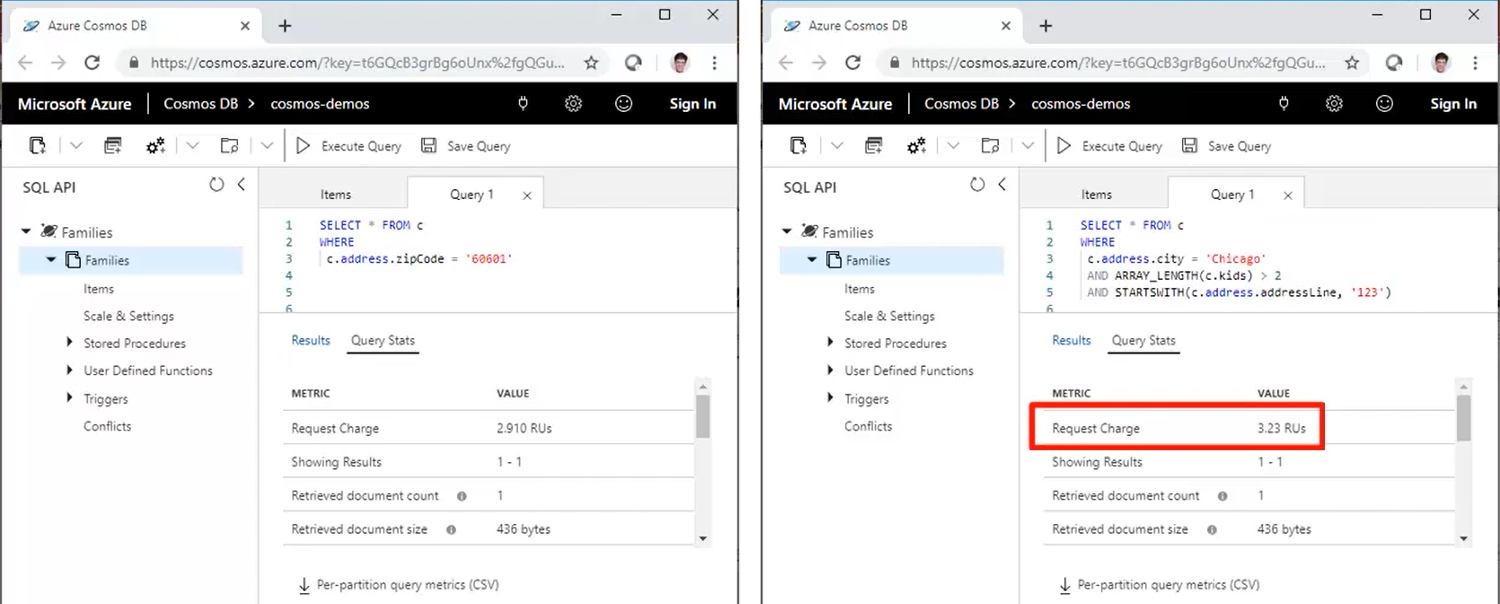


**Capacity Calculator:**

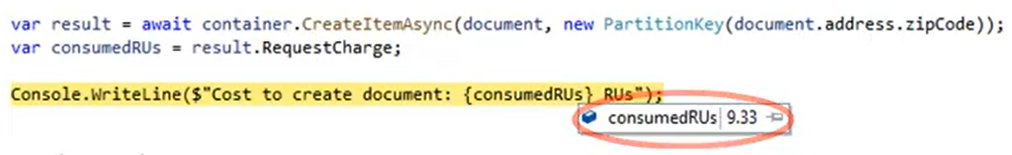
<https://cosmos.azure.com/capacitycalculator/>

Note: Sign-In to upload the document / JSON objects and get the correct estimates.

**Monitoring RU Consumption**



**In Code:**

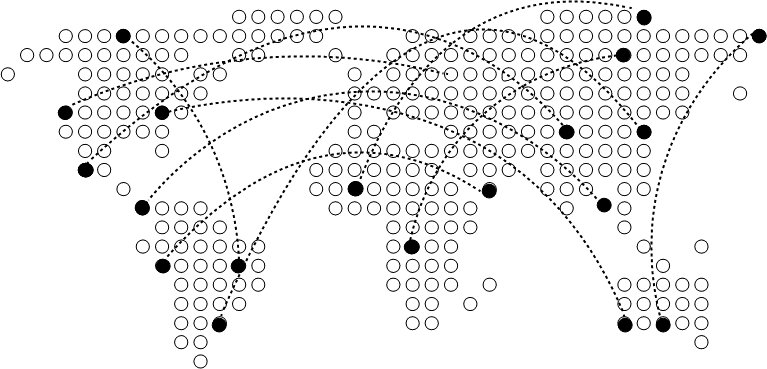


#### **Global replication and Consistency Levels**

Azure Cosmos DB has a feature referred to as ***turnkey global distribution*** that automatically replicates data to other Azure datacenters across the globe without the need to manually write code or build a replication infrastructure.

**Turnkey Global Distribution**: Regions can be dynamically added or removed.

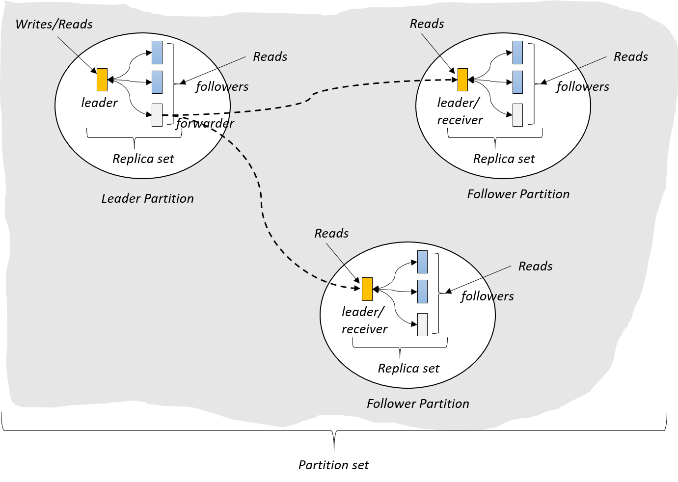
**Multi-master**: Enable write across all regions with automatic failover.



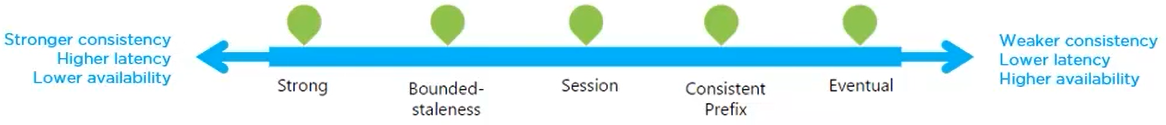
**Replication with in region:** Data moves very fast between neighboring racks (typically less than 1ms)

**Global Replication:** It takes hundreds of milliseconds to move data across continents.

**Consistency levels:**

****

How to ensure that consistent read across replicas:



* Each model provides availability and performance tradeoffs and is backed by the SLAs.
* The consistency levels are region-agnostic and are guaranteed for all operations regardless of the region from which the reads and writes are served, the number of regions associated with your Azure Cosmos account, or whether your account is configured with a single or multiple write regions.
* Read consistency applies to a single read operation scoped within a partition-key range or a logical partition. The read operation can be issued by a remote client or a stored procedure.

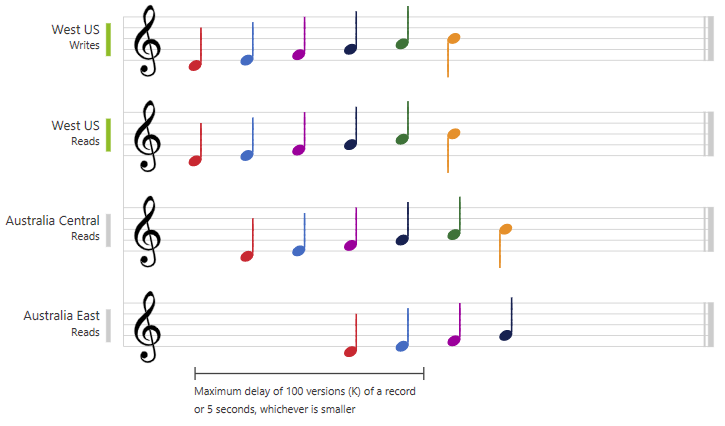
**Strong Consistency**

* When a write operation is performed on your primary database, the write operation is replicated to the replica instances. The write operation is committed (and visible) on the primary only after it has been committed and confirmed by all replicas.
* Strong consistency offers a linearizability guarantee. Linearizability refers to serving requests concurrently. The reads are guaranteed to return the most recent committed version of an item. A client never sees an uncommitted or partial write. Users are always guaranteed to read the latest committed write.
* No dirty reads but write latency increases.



**Bounded Staleness**

* This level is similar to the Strong level with the major difference that you can configure **how stale** documents can be within replicas. Staleness refers to the quantity of time (or the version count) a replica document can be behind the primary document.
* The reads are guaranteed to honor the consistent-prefix guarantee.
* The reads might lag behind writes by at most “K” versions (i.e., "updates") of an item or by “T” time interval. In other words, when you choose bounded staleness, the "staleness" can be configured in two ways:
  + The number of versions (K) of the item
  + The time interval (T) by which the reads might lag behind the writes
* Dirty reads possible (if read with T time or K versions)



**[Session]**

* This level guarantees that all read and write operations are consistent within a user session.
* Within the user session, all reads and writes are monotonic and guaranteed to be consistent across primary and replica instances.
* Clients outside of the session performing writes will see consistent prefix consistency.
* No dirty reads for writers (read your own writes) but Dirty read possible for other users.



**Consistent Prefix**

* This level has loose consistency but guarantees that when updates show up in replicas, they will show up in the correct order (that is, as prefixes of other updates) without any gaps.
* Consistent prefix consistency level guarantees that reads never see out-of-order writes.
* Dirty read possible.
* Examples include showing threaded comments.



**Eventual**

* This level has the loosest consistency and essentially commits any write operation against the primary immediately. Replica transactions are asynchronously handled and will eventually (over time) be consistent with the primary.
* This tier has the best performance, because the primary database does not need to wait for replicas to commit to finalize its transactions.
* There's no ordering guarantee for reads. In the absence of any further writes, the replicas eventually converge.
* Dirty reads possible.
* Examples include count of Retweets, Likes or non-threaded comments.



|  |  |
| --- | --- |
| **Consistency Level** | **Guarantees** |
| Strong | Linearizable reads…No Dirty Reads |
| Bounded Staleness | Consistent Prefix. Reads lag behind writes by k prefixes or t interval |
| Session | Consistent Prefix. Monotonic reads, monotonic writes, read-your-writes, write-follows-reads |
| Consistent Prefix | Updates returned are some prefix of all the updates, with no gaps |
| Eventual | Dirtry Reads |

**Session Consistency:** Session is controlled using a “session token”.

* Session tokens are automatically cached by the Client SDK
* Can be pulled out and used to override other requests (to preserve session between multiple clients)

**Existing: { id = "1", value = "X" }**

===============================================

string sessionToken;

using (DocumentClient client = new DocumentClient(new Uri(""), "")) **//East US**

{

ResourceResponse<Document> response = client.UpsertDocumentAsync(

collectionLink,

new { id = "1", value = "Y" }

).Result;

ResourceResponse<Document> read = client.ReadDocumentAsync( **//East US**

documentLink

).Result; **//Returns Y**

sessionToken = response.SessionToken;

}

=================================================

using (DocumentClient client = new DocumentClient(new Uri(""), "")) **//Australia East**

{

ResourceResponse<Document> read = client.ReadDocumentAsync(

documentLink,

new RequestOptions { SessionToken = sessionToken }

).Result; **//Returns Y**

}

=================================================

using (DocumentClient client = new DocumentClient(new Uri(""), "")) **//Central India**

{

ResourceResponse<Document> read = client.ReadDocumentAsync(

documentLink,

).Result; **//Returns X Immediately but Y Eventually**

}

**Consistency can be relaxed on a per-request basis**

We can **override** the consistency at request level but it can only **weaken** the default consistency level.

client.ReadDocumentAsync(

documentLink,

new RequestOptions { ConsistencyLevel = ConsistencyLevel.Eventual }

);

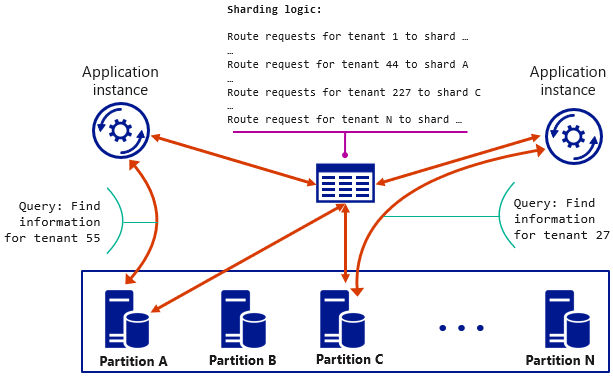
**Horizontal Scaling using Partitioning**

***Containers*** are logical resources and can span one or more physical partitions or servers. The number of partitions is determined by Azure Cosmos DB based on the storage size and throughput provisioned for a container or set of containers.

If you are already familiar with the sharding pattern, the idea of dynamic partitioning is not very different.

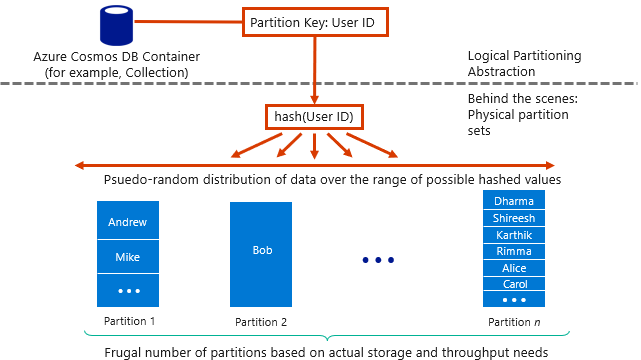
A ***physical partition*** is a fixed amount of reserved solid-state drive (SSD) back-end storage combined with a variable amount of compute resources (CPU and memory). Each physical partition is replicated for high availability.

* Partitioning will help us to massive scale our database not just for storage but also throughput.
* You can create one container and let it grow because internally it created multiple partitions.
* It’s a physical fixed capacity data buckets.
* Partition key values are hashed, hashed value determines the physical partition for storing each item.
* Partition keys are immutable. If you need to change your partition key, you should move your data to a new container with your new desired partition key.
* Physical Partitions can host multiple partition keys. Items with the same partition key value are physically stored together on the same partition.

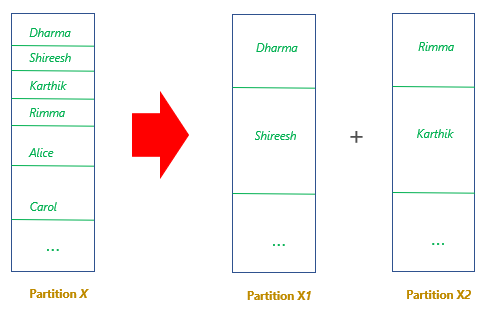


A ***logical partition*** is a partition within a physical partition that stores all the data associated with a single partition key value. Partition ranges can be dynamically subdivided to seamlessly grow the database as the application grows while simultaneously maintaining high availability. When a container meets the partitioning prerequisites, partitioning is completely transparent to your application. Azure Cosmos DB handles distributing data across physical and logical partitions and routing query requests to the right partition.

Transactions (in stored procedures or triggers) are allowed only against items in a single logical partition.



Cosmos DB automatically splits the partition to manage growth.



Partition management is completely taken care of by the system, you don’t have to lift a finger… the database takes care of you.

**Choosing the Right Partition Key**

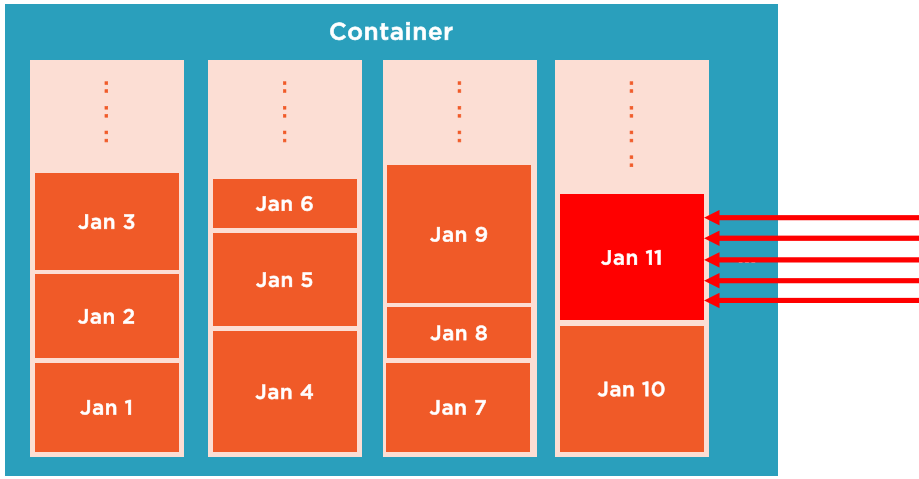
a) A single logical partition has an upper limit of 10 GB of storage.

**b)** Generally, writes should be distributed uniformly across partitions.

For example, user profile data with a user ID and creation date

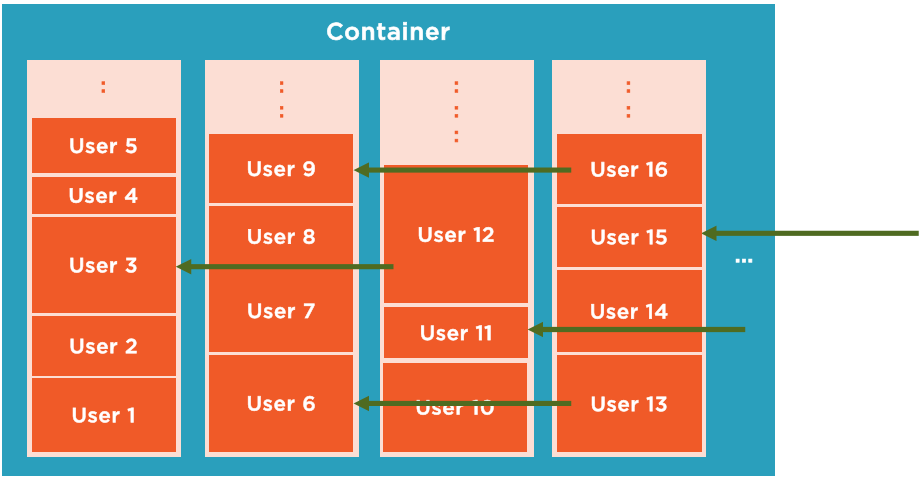
- Partitioning by creation date

Bad idea! All writes of the day are directed to the same partition.



- Partition by user ID

Much better! Writes are directed to different partitions per user



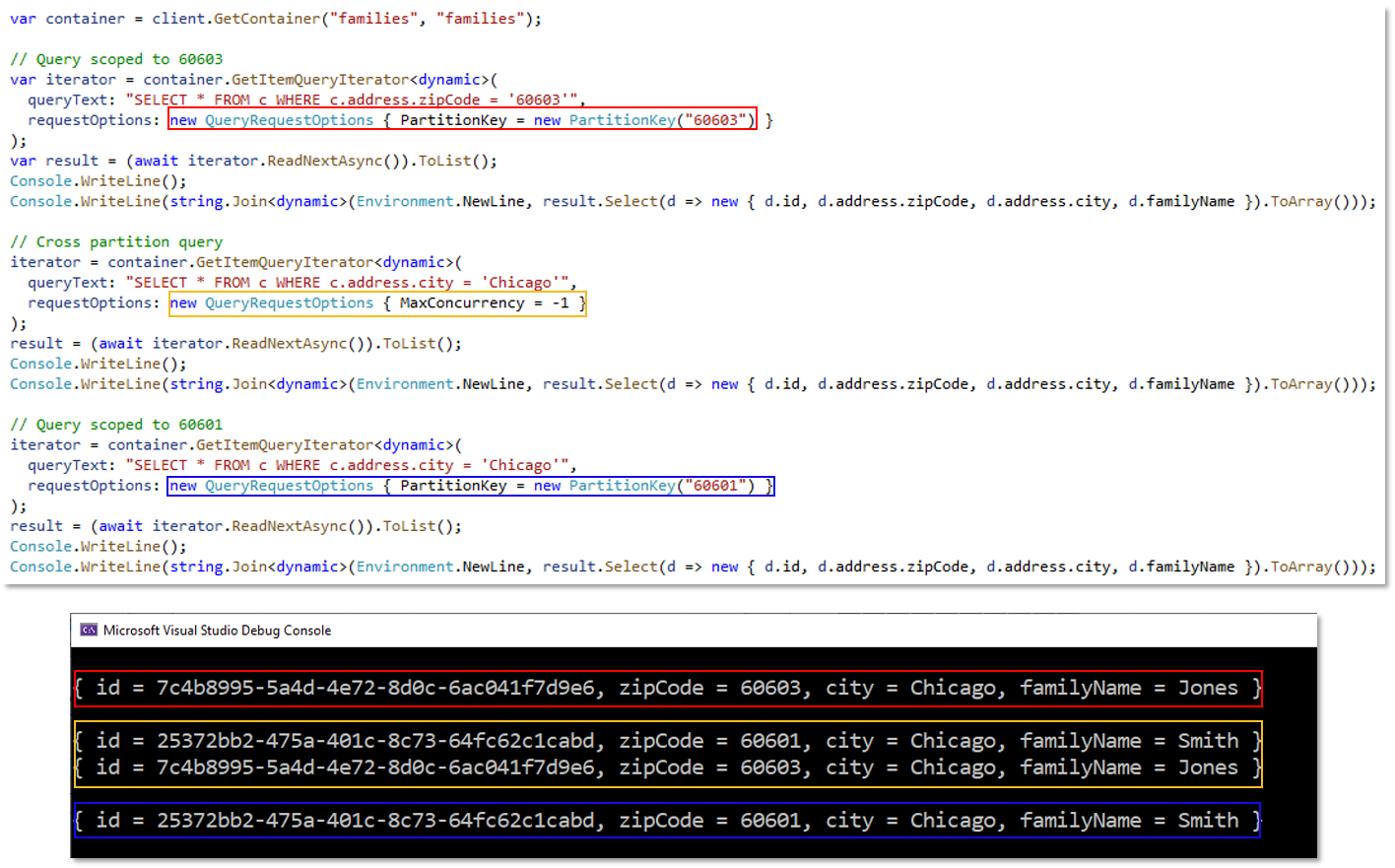
**c)** Create multiple containers for varying throughput needs (even though they have same partition key)

- Throughput is purchased at the container level

**Cross Partition Queries**

Candidates for partition keys might include properties that appear frequently as a filter in your queries. Queries can be efficiently routed by including the partition key in the filter predicate.





**NOTE:**

If your **container is small**, you probably don't have enough physical partitions to need to worry about the performance impact of cross-partition queries. Most small containers in Azure Cosmos DB only require **one or two physical** partitions.

If your container could grow to **more than a few physical** partitions, then you should make sure you pick a partition key that **minimizes** cross-partition queries.

Your container will require more than a few physical partitions when either of the following are true:

* Your container will have over 30,000 RU's provisioned.
* Your container will store over 100 GB of data.

**Concatenate multiple properties of an item:**

You can form a partition key by concatenating multiple property values into a single artificial partitionKey property. These keys are referred to as synthetic keys.

{

"deviceId": "abc-123",

"date": 2018

}

One option is to set /deviceId or /date as the partition key. Another option is to concatenate these two values into a **synthetic partitionKey** property that's used as the partition key

{

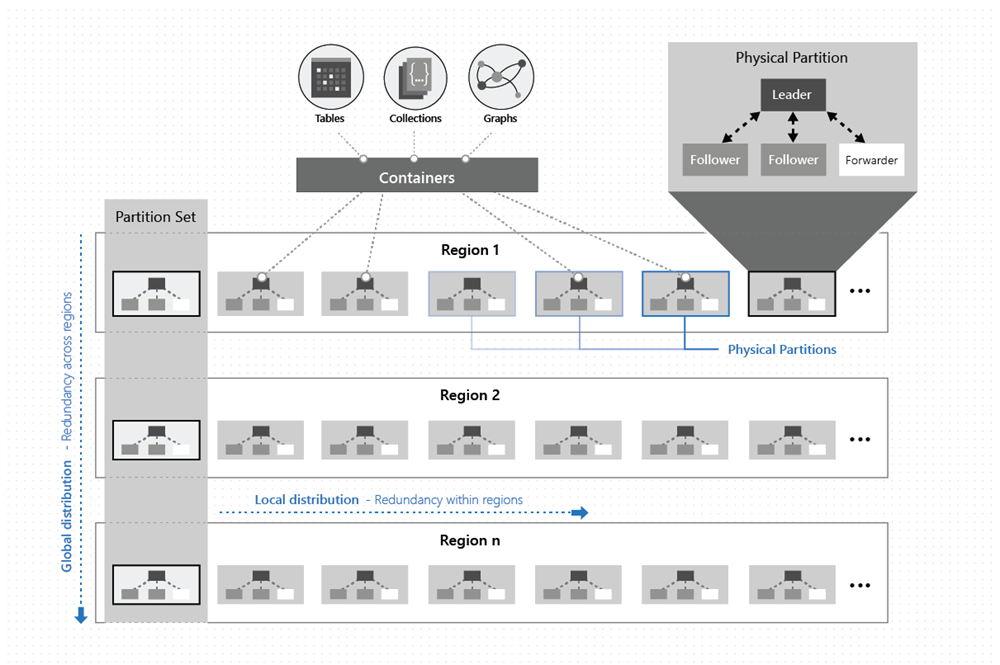
"deviceId": "abc-123",

"date": 2018,

**"myPartitionKey": "abc-123-2018"**

}

**High Availability with Cosmos DB**

**Diagram

Description automatically generated**

**Multi-region accounts with a single-write region (write region outage):**

* During a write region outage, the Cosmos account automatically promotes a secondary region to be the new primary write region.
* Any write data that was not replicated when the region failed, is made available through the conflicts feed.
* When the impacted write region recovers, it becomes automatically available as a read region.

**Multi-region accounts with a single-write region (read region outage):**

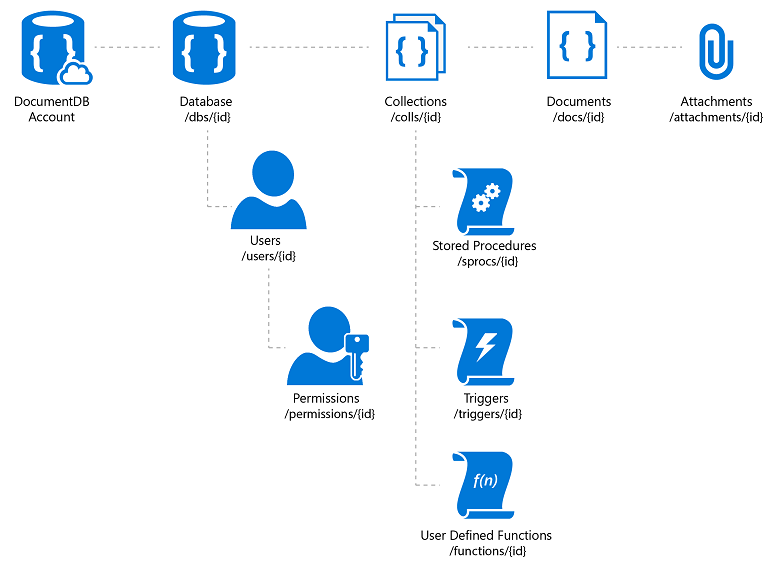
* During a read region outage, Cosmos accounts using any consistency level or strong consistency with three or more read regions will remain highly available for reads and writes.
* The impacted region is automatically disconnected and will be marked offline.
* If none of the regions in the preferred region list is available, calls automatically fall back to the current write region.

**High availability with Cosmos DB in the event of regional outages (Multi region write)**

* With Cosmos DB, before a write operation is acknowledged to the client, the data is durably committed by a quorum of replicas within the region that accepts the write operations.
* Multi-region accounts configured with multiple-write regions will be highly available for both writes and reads.

**Core (SQL) OR DocumentDB**

The image below shows the relationships between the resources:



**https://dssdemodb.documents.azure.com:443/dbs/<database-id>/colls/<col-id>/docs/<doc-id>**

**Addressing a resource**

All resources are URI addressable. The value of the **\_self** property of a resource represents the relative URI of the resource. The format of the URI consists of the /<feed>/{\_rid} path segments:

|  |  |
| --- | --- |
| **Value of the \_self** | **Description** |
| /dbs | Feed of databases under a database account |
| /dbs/{dbName}  UriFactory.CreateDatabaseUri(DatabaseId) | Database with an id matching the value {dbName} |
| /dbs/{dbName}/colls/ | Feed of collections under a database |
| /dbs/{dbName}/colls/{collName} | Collection with an id matching the value {collName} |
| /**dbs**/{dbName}/**colls**/{collName}/**docs** | Feed of documents under a collection |
| /dbs/{dbName}/colls/{collName}/docs/{docId} | Document with an id matching the value {doc} |
| /dbs/{dbName}/users/ | Feed of users under a database |
| /dbs/{dbName}/users/{userId} | User with an id matching the value {user} |
| /dbs/{dbName}/users/{userId}/permissions | Feed of permissions under a user |
| /dbs/{dbName}/users/{userId}/permissions/{permissionId} | Permission with an id matching the value {permission} |

Each resource has a unique user defined name exposed via the id property. The id is a user defined string, of up to 256 characters that is unique within the context of a specific parent resource.

Note: for documents, if the user does not specify an id, our supported SDKs will automatically generate a unique id for the document.

**Programming CosmosDB**

**Creating Document by using .NET**

using Microsoft.Azure.Cosmos;

using Microsoft.Azure.Cosmos.Linq;

**// Get container reference**

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

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

OR

Database database = await clien.**CreateDatabaseIfNotExistsAsync**(databaseName,  throughput: 10000);

string collectionName = "ExampleCollection";

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

**OR**

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

**// create anonymous type in .NET**

Product orangeSoda = new Product {

    id = "7cc3212d-0e2c-4a13-b348-f2d879c43342",

    name = "Orange Soda", group = "Beverages",

    diet = false, price = 1.50m, quantity = 2000

};

// Upload document – Create a new document  
Product item = await container.**CreateItemAsync**(orangeSoda);

//OR – Create or Replace document

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

**Reading documents by using .NET**

// Get container reference

CosmosClient client = new CosmosClient(endpoint, key);

Container container = client.GetContainer(databaseName, collectionName);

// Get unique fields  
string id = "7cc3212d-0e2c-4a13-b348-f2d879c43342";

PartitionKey **partitionKey** = new PartitionKey("Beverages");

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

partitionKey

);

//Deserialize response  
Product item = response.Resource;

**Query Documents using .NET**

// Get container reference

CosmosClient client = new CosmosClient(endpoint, key);

Container container = client.GetContainer(databaseName, collectionName);

**// Use SQL query language**

FeedIterator<Product> iteratorOld = container.**GetItemQueryIterator**<Product>(

    "SELECT \* FROM products p WHERE p.diet = false"

);  
//OR

**// Use LINQ query language**

FeedIterator<Product> iterator = container.**GetItemLinqQueryable**<Product>()

    .Where(p => !p.diet)

    .ToFeedIterator();

// Iterate over results

while (iterator.HasMoreResults)

{

    FeedResponse<Product> batch = await iterator.**ReadNextAsync**();

    foreach(Product item in batch)

    { }

}

**Optimistic Concurrency**

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

try

{

    var ac = new **AccessCondition** { Condition = readDoc.**ETag**, Type = AccessConditionType.**IfMatch** };

    await client.**ReplaceDocumentAsync**(readDoc, new **RequestOptions** {AccessCondition = ac });

}

catch (DocumentClientException dce)

{

    if (dce.StatusCode == HttpStatusCode.PreconditionFailed)

    {

        Console.WriteLine("Another process has updated the record");

    }

}

|  |  |
| --- | --- |
| {  "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" },  "creationDate": 1431620472,  "isRegistered": true  } | {  "id": "WakefieldFamily",  "parents": [  { "familyName": "Wakefield", "givenName": "Robin" },  { "familyName": "Miller", "givenName": "Ben" }  ],  "children": [  {  "familyName": "Merriam",  "givenName": "Jesse",  "gender": "female", "grade": 1,  "pets": [  { "givenName": "Goofy" },  { "givenName": "Shadow" }  ]  },  {  "familyName": "Miller",  "givenName": "Lisa",  "gender": "female",  "grade": 8  }  ],  "address": { "state": "NY", "county": "Manhattan", "city": "NY" },  "creationDate": 1431620462,  "isRegistered": false  } |

**More Queries and Results:**

SELECT \* FROM Families f WHERE f.id = "AndersenFamily"

Fetches only one Entity in array.

|  |  |
| --- | --- |
| SELECT {"Name":f.id, "City":f.address.city} **AS Family**  FROM Families f  WHERE f.address.city = f.address.state | [{  "Family": {  "Name": "WakefieldFamily",  "City": "NY"  }  }] |
| SELECT c.givenName  FROM Families f  JOIN c IN f.children  WHERE f.id = 'WakefieldFamily'  ORDER BY f.address.city ASC | [  { "givenName": "Jesse" },  { "givenName": "Lisa"}  ] |
| SELECT f.**address**  FROM Families f  WHERE f.id = "AndersenFamily" | [{  "**address**": {  "state": "WA",  "county": "King",  "city": "seattle"  }  }] |
| SELECT f.address.**state**, f.address.**city**  FROM Families f  WHERE f.id = "AndersenFamily" | [{  "state": "WA",  "city": "seattle"  }] |

More About Queries: <https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-sql-query>