Agenda: Azure Cosmos DB Service

- What is Cosmos DB?
    - Understanding NoSQL
    - CosmosDB Introduction
- Creating Cosmos Db Account using Azure Portal.
- Create Database and Container
- Create Document
- Query Document
- Management Options
- Auto Indexing
- Managing Throughput using RU's
- Horizontal Scaling using Partitioning.

What is Cosmos DB?

What is NoSQL Database?

- Non-Relational Database management system.
- Store data differently than relational table.
- Provides flexibility in schema
- Designed for Scale out.Comes in variety of data models

SQL	NoSQL
Data stored as Tables Tables fixed Schema(Col,Datatype,Constraints) Can create Relationship	Many implementations like Key-value,Graph,Document, Flexible Schema
Storage is concentrated	Storage is partitioned based on Hash
Vertical Scalability	Horizontal Scaling.
SQL language is used and Queries are flexible	Vendor specific languages and Rest API are used and Queries are less flexible
Small project+Low scale+Unknown access pattern Large project + High Scale+Relational queries(SQL with read replica)	Medium large project+Hogh scale+High performance

Examples:

- **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.There is no Schema management.Without downtime Schema changes can be replicated to 100's of nodes globally.

What is Cosmos DB

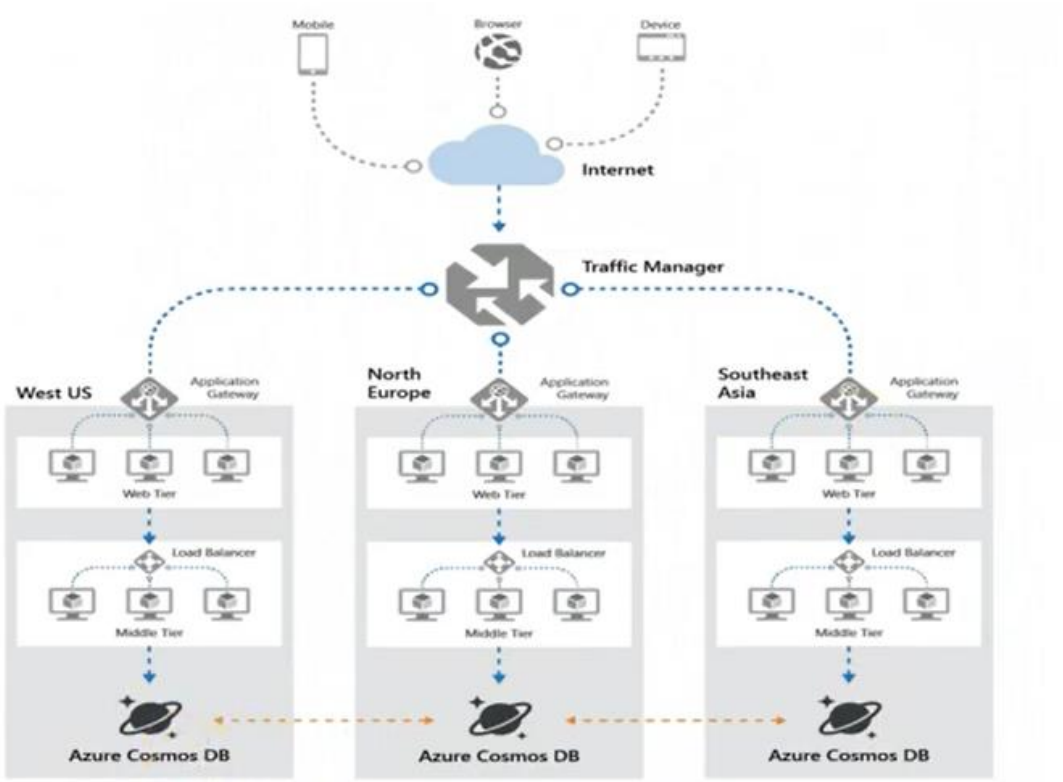
- It's an Evolution of Document DB which was publicly available from 2015. Document DB is now referred as SQL API and is just one of the API's supported in Cosmos DB
- Introduced in the year 2017.
- Azure Cosmos DB is **Microsoft's globally distributed, multi-model/multi-api database**.
- Azure Cosmos DB enables you to elastically and independently **scale throughput and storage** across any number of Azure's geographic regions. So it has virtually unlimited Scale.
- You can have app with few GB of storage and few hundred of request which can be scaled to PB's of storage and millions of request.This is achieved with **Horizontal Scaling**.
- **It offers throughput, latency, availability, and consistency guarantees** with comprehensive service level agreements (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.

Capability Comparison:

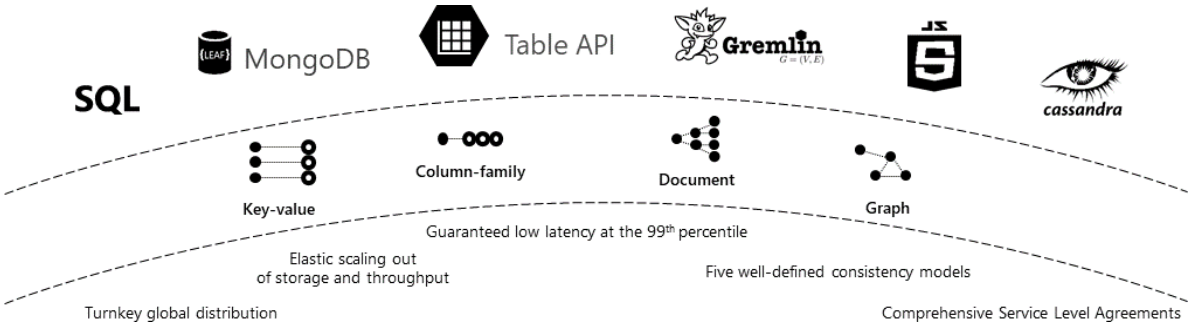
Azure Cosmos DB provides the best capabilities of relational and non-relational databases.

Capabilities	Relational databases	Non-relational (NoSQL) databases	Azure Cosmos DB
Global distribution	No	No	Yes, turnkey distribution in 30+ regions, with multi-homing APIs
Horizontal scale	No	Yes	Yes, you can independently scale storage and throughput
Latency guarantees	No	Yes	Yes, 99% of reads in <10 ms and writes in <15 ms
High availability	No	Yes	Yes, Azure Cosmos DB is always on, has well-defined PACELC tradeoffs, and offers automatic and manual failover options
Data model + API	Relational + SQL	Multi-model + OSS API	Multi-model + SQL + OSS API
SLAs	Yes	No	Yes, comprehensive SLAs for latency, throughput, consistency, availability



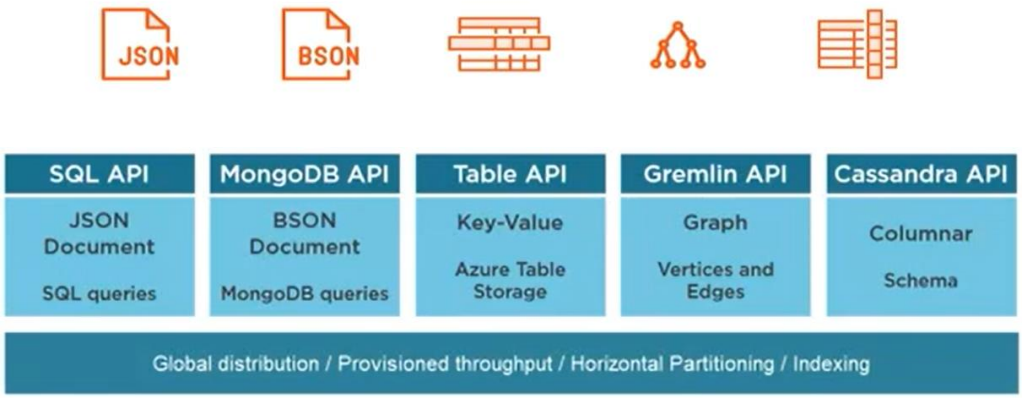
Multiple data models and popular APIs for accessing and querying data

- 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)**: A schema-less JSON database engine with rich SQL querying capabilities.
  2. **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.
  3. **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.

4. **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 Gremlin APIs. It stores entities which are called Nodes and Edges.
5. **Cassandra API**: A globally distributed Cassandra-as-a-Service powered by Azure Cosmos DB platform. Compatible with existing Apache Cassandra libraries, drivers, tools, and applications.



SQL DB database

- 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.

Key	Document

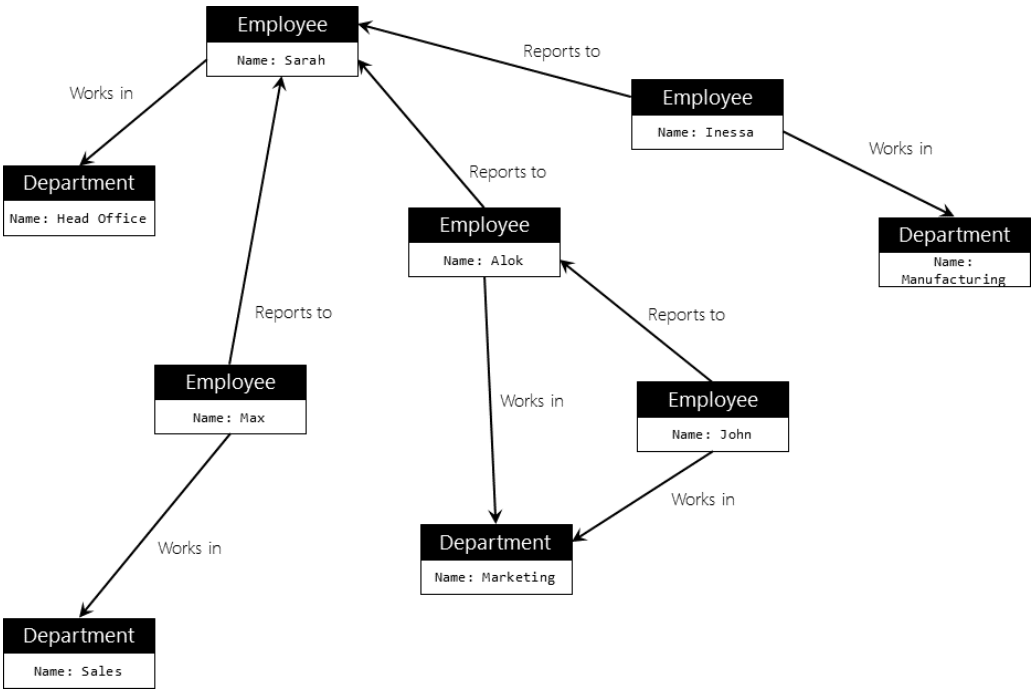
1001	<pre>[
	  {
	  "customerId": 344,
	  "orderItems": [
	  {
	  "productId": 4524,
	  "quantity": 1,
	  "price": 125.67
	  },
	  {
	  "productId": 3311,
	  "quantity": 4,
	  "price": 73.06
	  }
	  ],
	  "orderDate": "2017-10-18T12:27:30 +04:00"
	  }
	]</pre>

1002	<pre>[
	  {
	  "customerId": 263,
	  "orderItems": [
	  {
	  "productId": 4076,
	  "quantity": 3,
	  "price": 257.64
	  }
	  ],
	  "orderDate": "2014-01-31T02:09:02 +05:00"
	  }
	]</pre>

1003	<pre>[
	  {
	  "customerId": 308,
	  "orderItems": [
	  {
	  "productId": 1957,
	  "quantity": 1,
	  "price": 279.63
	  }
	  ],
	  "orderDate": "2016-09-18T01:33:53 +04:00"
	  }
	]</pre>

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.



CosmosDb Use Cases:

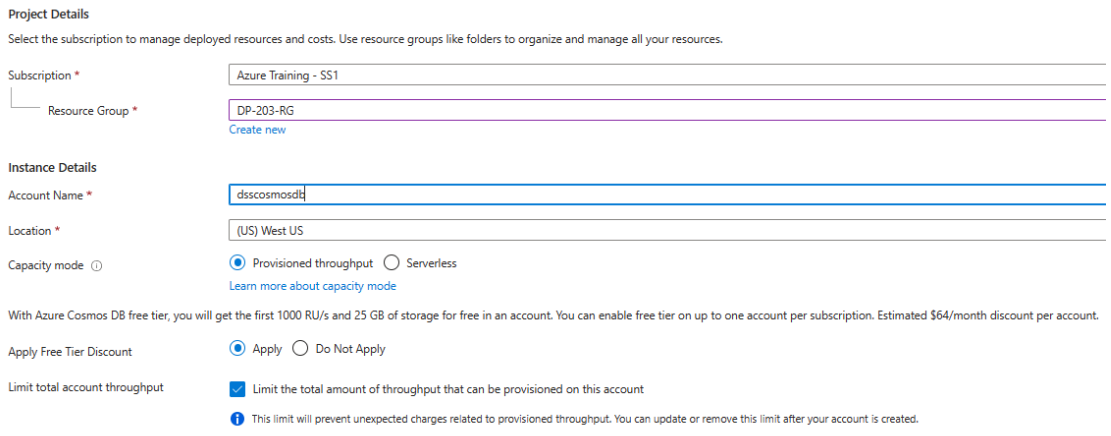
Retail Industry (Catlog Data)

Catalog data are user accounts, product catalogs, IoT device registries, and bill of materials systems.

Creating Cosmos DB Account using Portal

Lab1:Create CosmosDB Account

1. Azure Portal → Search → Azure Cosmos DB→+Create→Select Azure Cosmos DB for NoSQL →create



→Keep all default options→Review+Create

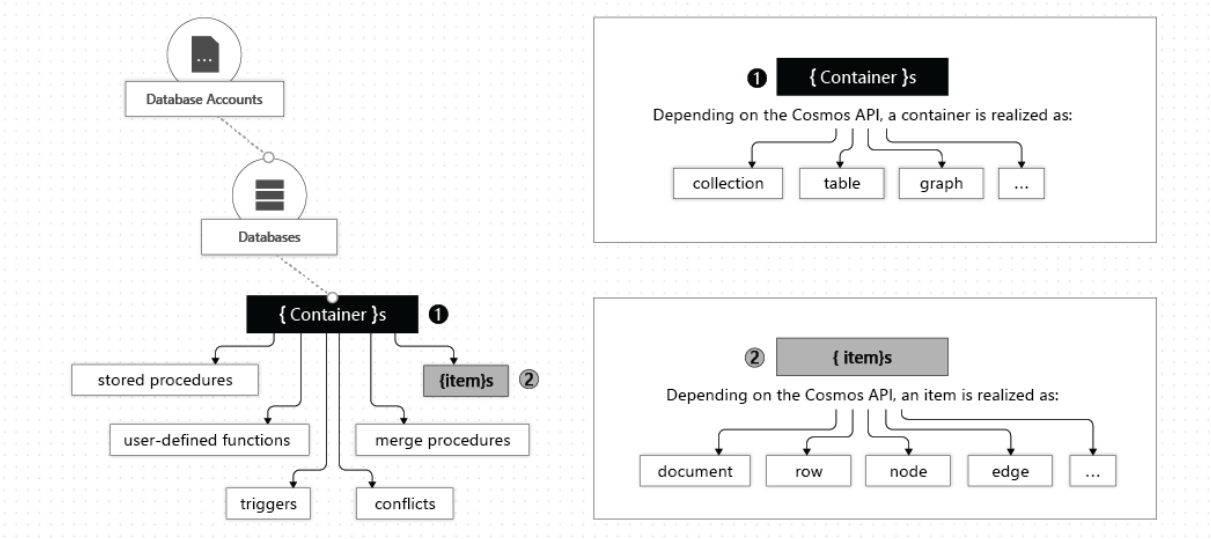
Create Database and Container

Azure Cosmos databases

- You can create one or multiple Azure Cosmos databases under your account.
- A database is analogous to a namespace.
- A database is the unit of management for a set of Azure Cosmos containers.

Azure Cosmos containers

- An Azure Cosmos container is the unit of scalability both for provisioned throughput and storage.
- A container is horizontally partitioned and then replicated across multiple regions.
- The items that you add to the container and the throughput that you provision on it are automatically distributed across a set of logical partitions based on the partition key.

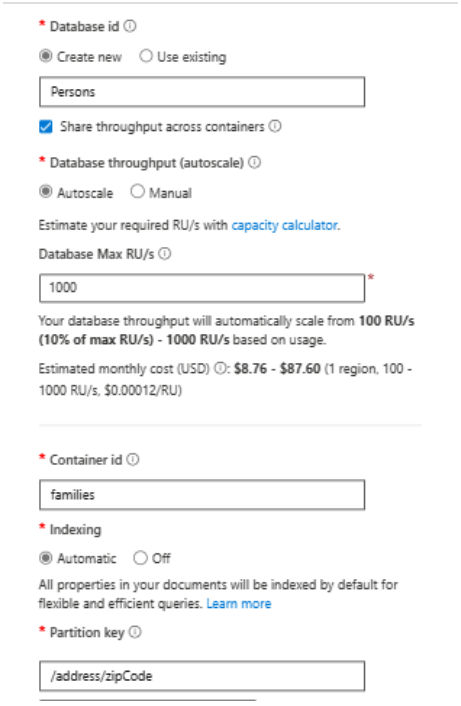


A container is specialized into API-specific entities as shown in the following table:

Azure Cosmos entity	SQL API	Cassandra API	Azure Cosmos DB API for MongoDB	Gremlin API	Table API
Azure Cosmos container	Container	Table	Collection	Graph	Table
Azure Cosmos item	Document	Row	Document	Node OR Edge	Item

Lab2:Create Database and Container

Overview→+Add Container→+ New Container



→OK

Lab3:Add Sample Data and Query

families→item→+New Item→Use following document and add items one by one→save

Sample Families Document

```
{
	"familyName":"Smith",
	"address":{
		"addressLine":"123 Main Street",
		"city":"Chicago",
		"state":"IL",
		"zipCode":"60601"
	},
	"parents":[
		"Peter",
		"Alice"
	],
	"kids":[
		"Adam",
		"Jacqueline",
		"Joshua"
	]
}
```

```
{
    "familyName":"Jones",
    "address":{
        "addressLine":"456 Harbor Boulevard",
        "city":"Chicago",
        "state":"IL",
        "zipCode":"60603"
```

```
    },
    "parents":[
        "David",
        "Diana"
    ],
    "kids":[
        "Evan"
    ],
    "pets":[
        "Lint"
    ]
}
```

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

Cosmos DB Local Emulator

- Emulate Cosmos DB in local development environment.It is Identical to Cosmos DB .You do not require Azure Subscription,Cosmos DB account or internet connection.
- You can Develop and test locally without incurring any cost and finally deploy to Cloud.

http://aka.ms/cosmosdb-emulator

and open the explorer in browser

CosmodDB Management Options:

- Portal
- CLI
- PowerShell
- Notebook
- REST API

Query Document

Lab4:Query families container

1. Data Explorer→Click on  (New SQL Query Tab) and use following queries

```
SELECT * FROM c
WHERE STARTSWITH(c.address.addressLine,'123')


SELECT * FROM c
WHERE c.address.zipCode="60601"

SELECT * FROM c
WHERE ARRAY_LENGTH(c.kids)>2

SELECT * FROM c
WHERE IS_DEFINED(c.pets)
```

```
SELECT * FROM c
WHERE c.address.city="Chicago"
```

Refer: https://docs.microsoft.com/en-us/azure/cosmos-db/sql-query-getting-started

Auto Indexing

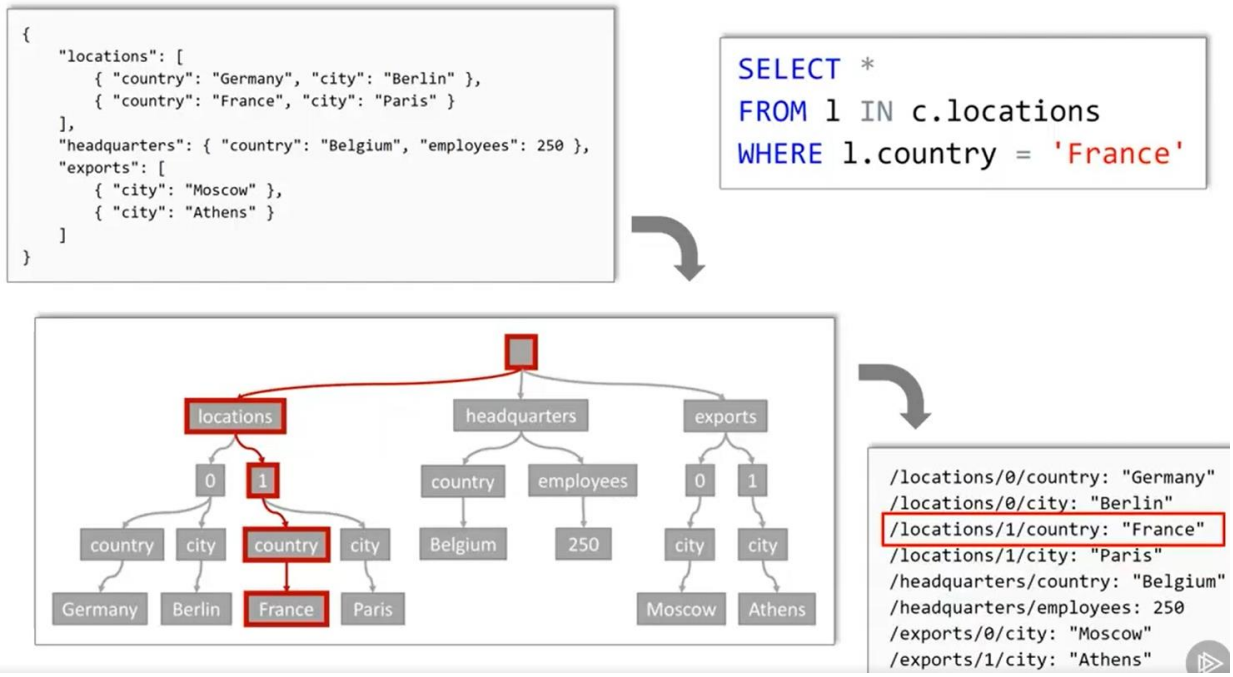
Cosmos Db by default index every property in every item.

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.

It is created in format of nodes and Leaf.Leaf is value.



Managing Throughput

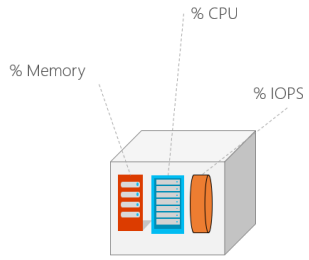
Performance can be measured in two ways

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.

Understanding 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.



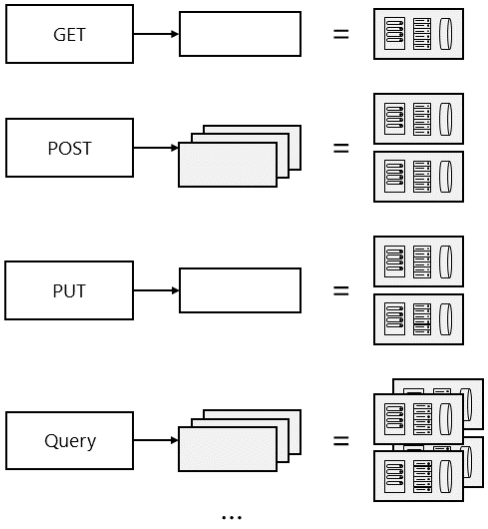
- You **reserve** several guaranteed request units to be available to your application on a per-second basis.
- Each operation in Azure Cosmos DB, including writing a document, performing a query, and updating a document, consumes CPU, memory, and Input/output operations per second (IOPS). That is, each 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.

Normalized across various access methods

1 RU = 1 read of 1 KB document

Each request consumes fixed RUs

Applies to reads, writes, queries, and stored procedure execution



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

Estimating Throughput Needs

Item size	Reads/second	Writes/second	Request units
1 KB	500	100	(500 * 1) + (100 * 5) = 1,000 RU/s
1 KB	500	500	(500 * 1) + (500 * 5) = 3,000 RU/s
4 KB	500	100	(500 * 1.3) + (100 * 7) = 1,350 RU/s
4 KB	500	500	(500 * 1.3) + (500 * 7) = 4,150 RU/s
64 KB	500	100	(500 * 10) + (100 * 48) = 9,800 RU/s
64 KB	500	500	(500 * 10) + (500 * 48) = 29,000 RU/s

* Based on Session consistency indexing policy set to None.

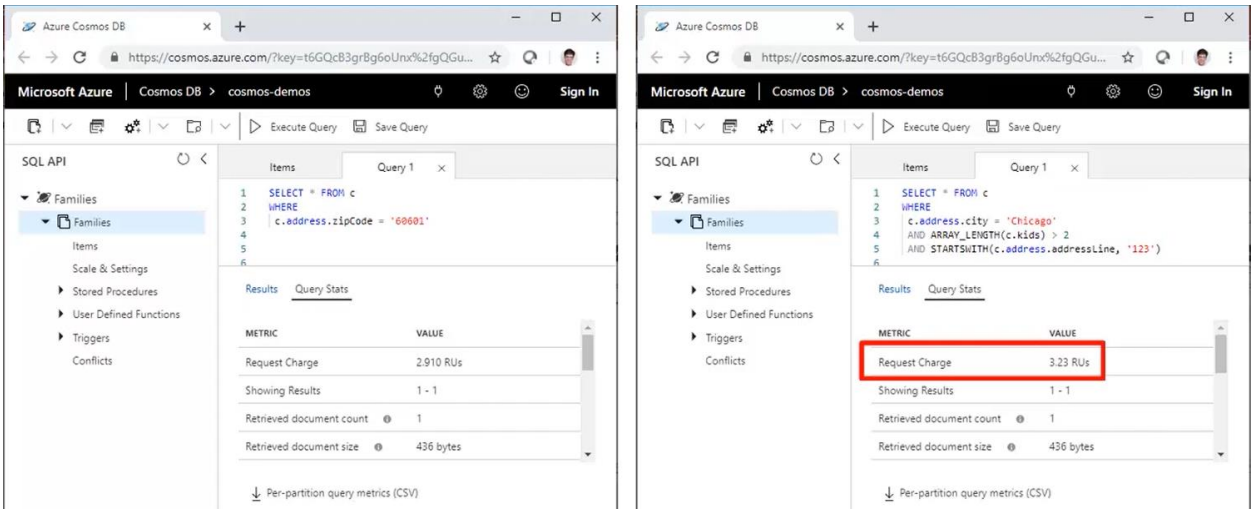
Capacity Calculator:

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

Pricing can be calculated based on SSD storage and Throughput.

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

Monitoring RU Consumption



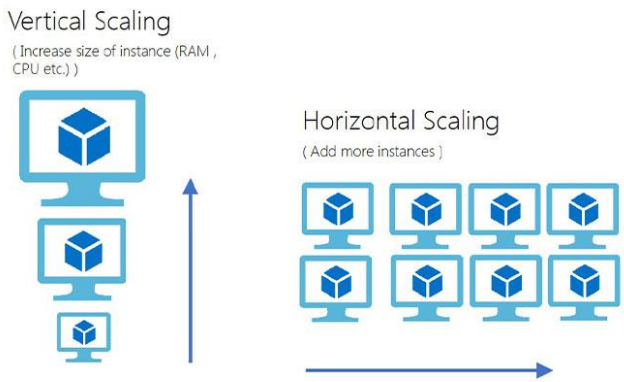
In Code:

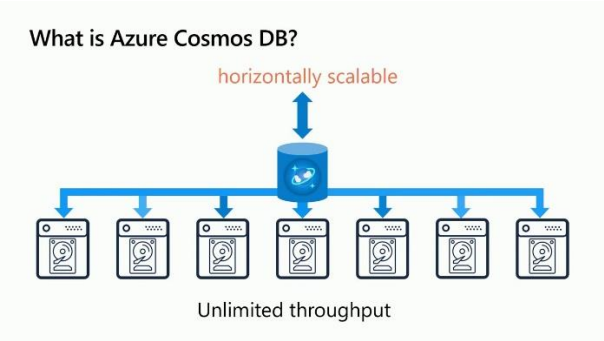
```
var result = await container.CreateItemAsync(document, new PartitionKey(document.address.zipCode));
var consumedRUs = result.RequestCharge;

Console.WriteLine($"Cost to create document: {consumedRUs} RUs");
```

consumedRUs 9.33

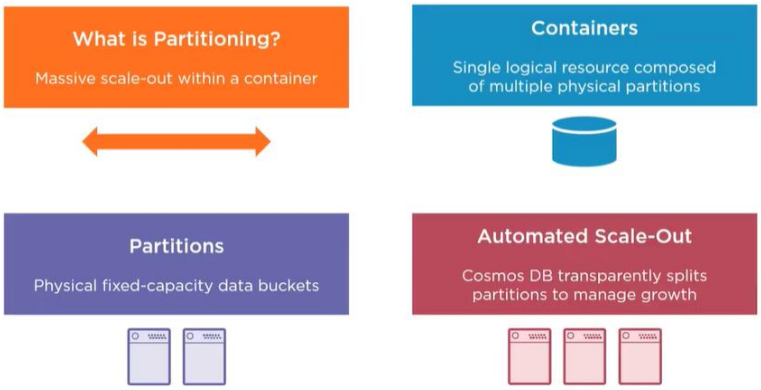
Horizontal Scaling using Partitioning

Vertical vs Horizontal Scaling:



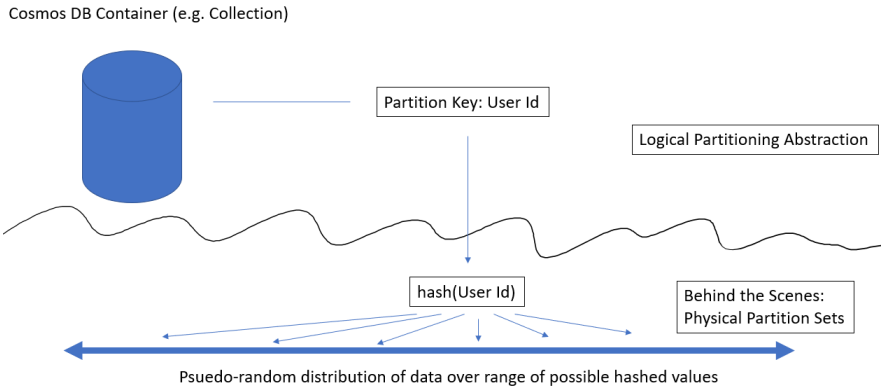


Partitioning:



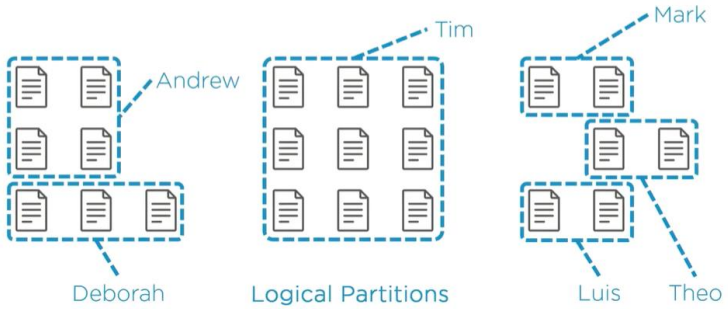
- Partitioning will help us to massive scale our database not just for storage but also throughput.
- You can create one container and let it grow because internally it created multiple partitions.
- It's a physical fixed capacity data buckets.
- Partition key values are hashed, hashed value determines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

- Cosmos DB transperantly **splits partitions** to manage the growth.



Logical partitioning:

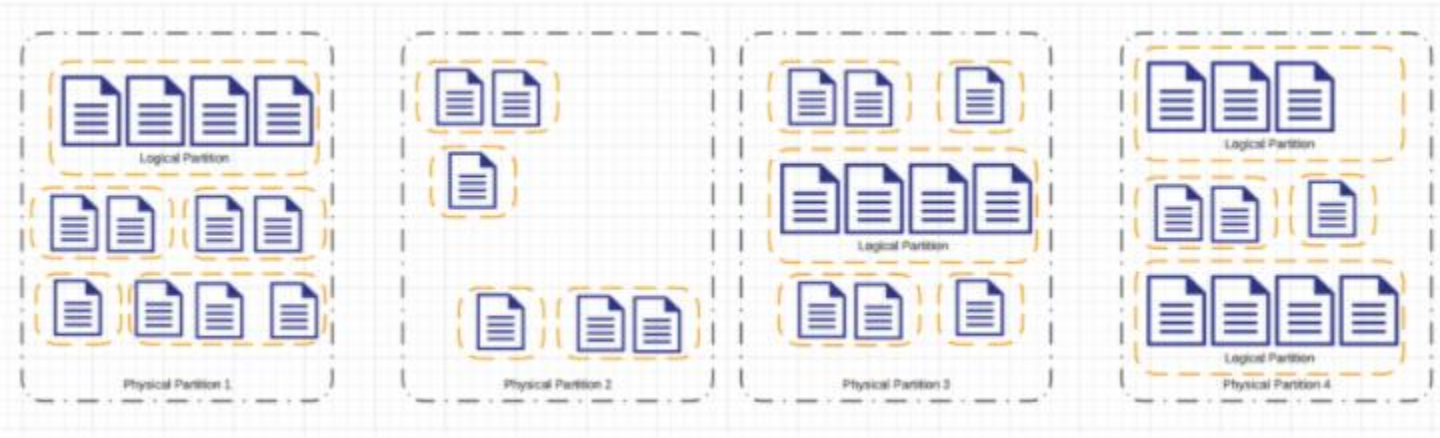
- A logical partition consists of a set of items that have the same partition key.
- There is no limit to the number of logical partitions in your container. Each logical partition can store up to 20GB of data



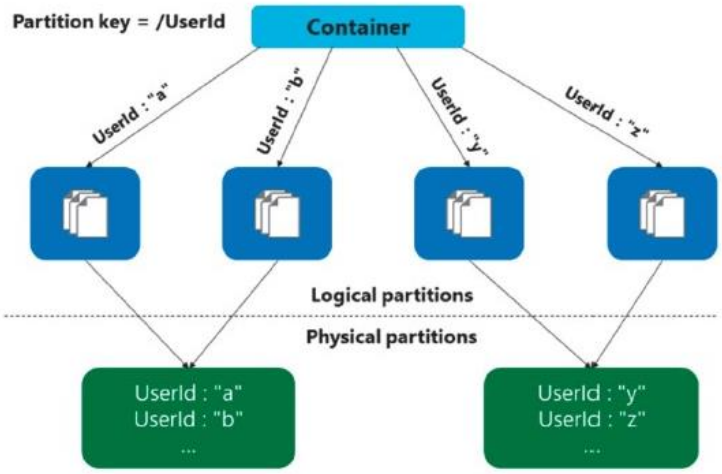
Physical Partition:

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

- A container is scaled by distributing data and throughput across physical partitions. Internally, one or more logical partitions are mapped to a single physical partition.
- Unlike logical partitions, physical partitions are an internal implementation of the system and they are entirely managed by Azure Cosmos DB.
- Each individual physical partition can store up to **50GB** data



Logical and Physical Partition Mapping:



Note: Max document size:2MB,Max logical partition size:20GB

Choosing the Right Partition Key

A. For all containers, your partition key should:

- Be a property that has a value which does not change. If a property is your partition key, you can't update that property's value.
- Have a high cardinality i.e. have a wide range of possible values.
- Spread request unit (RU) consumption and data storage evenly across all logical partitions. This ensures even RU consumption and storage distribution across your physical partitions. wide range of possible values ensures that the container is able to scale.

B. **For large read-heavy containers,Partition key based on Data Access pattern**:

- Choose propery that groups commonly queried/updated items together
- Generally, writes should be distributed uniformly across partitions.

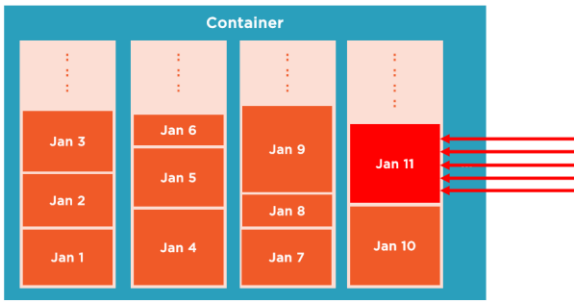
For example, In typically social media application,storing user profile data data with a user ID and creation date

- Partitioning by creation date

From storage perspective ,you get nice uniformity using date as partition key.But for write operation it is problem.

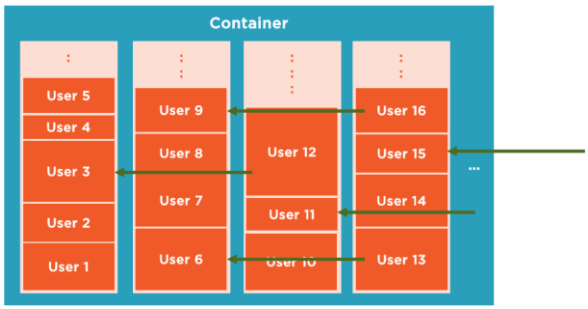
Bad idea! All writes of the day are directed to the same partition.

Throughput provisioned for a container is divided evenly among physical partitions.



- Partition by user ID

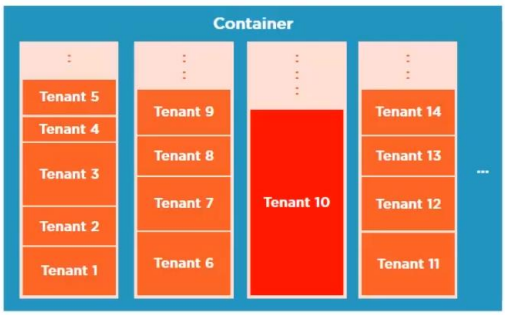
Much better! Writes are directed to different partitions per user



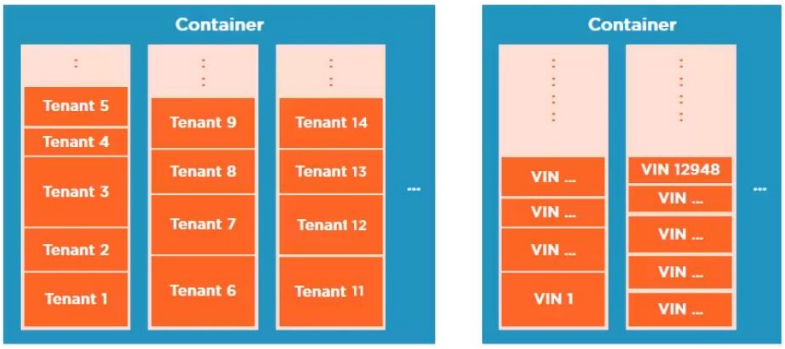
C. Create multiple containers for varying throughput needs

For Example:Multi tenant architecture with tenantId as Partition Key

Uneven distribution of storage and throughput



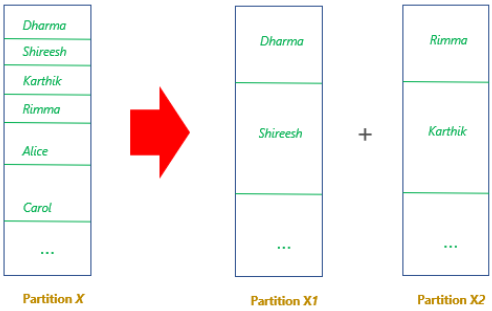
-Separate container for Tenant10



- Throughput is purchased at the container level

Partition Split:

- There is no limit to the total number of physical partitions in your container.
- As your provisioned throughput or data size grows, Azure Cosmos DB will automatically create new physical partitions by splitting existing ones.
- A physical partition split simply creates a new mapping of logical partitions to physical partitions.
- 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.