**Azure\_CosmosDb**

**Concepts of cosmosdb**

* Introduction
* Throughput and cost
* Horizantal Partitioning
* Global Distribution

**How-to**

* Data modeling and migration
* querying with SQL
* programming with .net SDK
* Building cosmosdb applications (serverside progeamming,batch & bulk operations , change feed)
* alternative way of modeling data in cosmosdb
* Table API > keyvalue store
* Gramlin API >graph db

**NoSQL ?**

* to solve problem of Big data. ie, 3 Vs
  + volume > db size Tb,Pb,..
  + velocity >Throughput , keep pace with requests that coming in fast (distributed workload across machines )
  + variety > this is about schema and problems that occur in relational DB regarding schema changes. as schemas evolve more, it is difficult to apply those schema changes to mission critical databases that require continuous uptime around clock.this is even more difficult for distributed DBs when those schema changes need to be deployed to multiple regions.
* Relation Dbs can't handle Big data
* in NoSql instead of one big machine (Tb,Pb size), scale-out to multiple machines in the face of volume , velocity and variety

**What is NoSql DB?**

* Distributed :: Replicas ensure high availability and resilience
  + NoSql db is ditributed from groundup.no extra effort needed.
  + multiple copies of data are distributed to separate replicas. more copies you have of same data , more available it is to more users. so, more resilient to failure if and when replica should go down.
* Scale-out :: Horizantal Partitoning for elastic storage and throughput.
  + horizontal partitioning, which balances the workload uniformly across a cluster of machines.these capabilities are built in.
* Schema-free :: no enforced schema
  + it doesn't mean there isn't any schema, but no enforced schema. So, every item in database have schema, which is its shape, but that shape can change without friction and downtime unlike relational databases and enforced schemas.

**Cosmos DB**

* Cosmos DB is massively scalable NoSQL database that's available to your application as a fully managed PaaS that runs on Azure.
* There are comprehensive service level agreements (SLAs) on **throughput, latency, consistency, and availability,** 
  + guaranteed no more than 52.56 minutes of downtime per year max with 99.99% availability
  + or no more than 5.26 minutes of downtime per year max with 99.999% availability
  + this you can get if you globally distribute your data.
* it delivers blazing fast reads and writes, typically within 10 ms or less, whether you've GB, TB or PB of data.
* Cosmos DB **automatically** manages the growth of your data by **horizontally partitioning** the workload behind the scenes.
* The process is transparent and provides elastic scale for virtually unlimited storage to handle volume, as well as throughput for velocity.
* Cosmos DB runs in Microsoft data centers throughout the world. Within any one data center, Cosmos DB **replicates your data automatically**, and that ensures high availability within that one region.
* you can also replicate your data across **multiple data centers** and distribute your data globally.
* Simply point and click on a map, and Cosmos DB does the rest
* You can also enable **multi‑master**, and that will ensure high performance writes, as well as reads in any region where your data is distributed.
* Cosmos DB has multiple APIs to support variety of schema‑free data models.
  + JSON documents using SQL API or the MongoDB API
  + Table data model> Table API
  + Graph data model > Gremlin API
  + columnar data model > Cassandra API

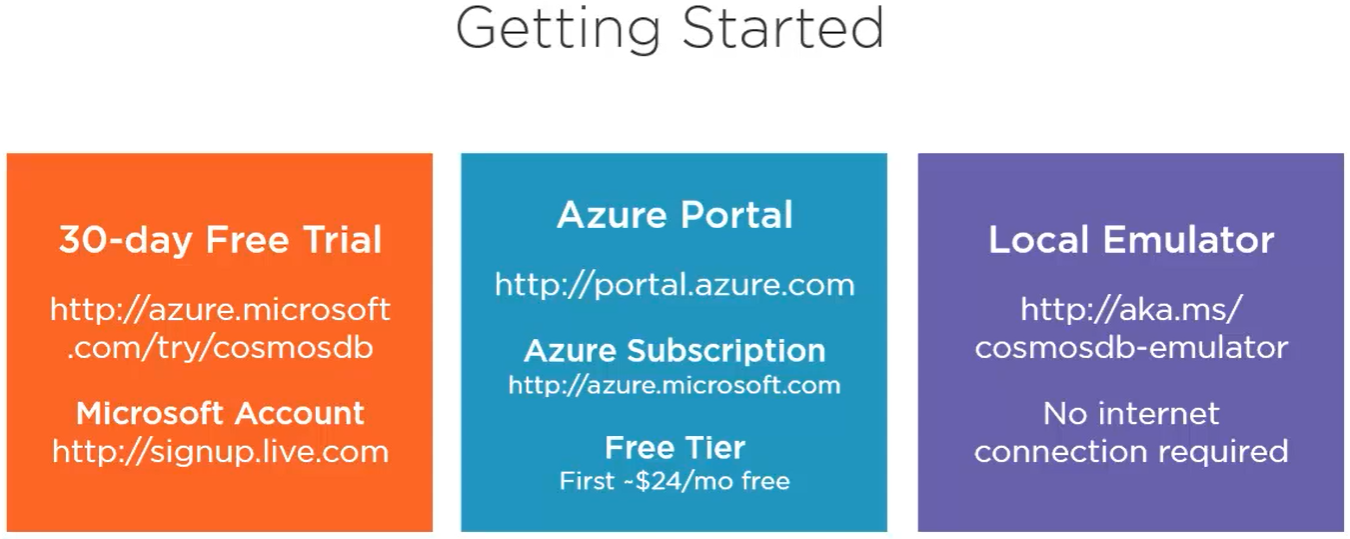
A picture containing diagram

Description automatically generated

Diagram

Description automatically generated

**Getting Started**



* above are 3 ways to use cosmosDb. Local emulator doesn’t support global distribution.

**Create CosmosDb:**

* portal.azure.com >create resource >search ‘azure cosmos db’ > Create ‘Core SQL API’
* **Core SQL** :: it gives json data model that can be queried using sql.it is special version of sql used to query json documents. This is most capable than others like mongo, table, Cassandra, gremlin APIs
* **Geo-redundancy** :: enables geo-replication between two regions
* **Mutli-region writes** :: enables both regions for writes and reads
* ‘Notebooks’ are not available for some regions. So some options might be disabled
* **Availablity zones** :: provides added redundancy for partial failures or outages within a single data center. This is useful for those data centres located in countries with data sovereignty and residency laws like Singapore, which does not permit geo-replication between southeast asia and hong kong. By using availability zones in south-east asia cosmosDb can offer redundancy in the event of major failure even within just that one region
* **Networking** ::
  + All networks :: our account is accessible over public internet
  + Public endpoint :: limited accessibility over selected networks
  + Private endpoint :: account can be accessible to only by clients within same virtual network or VNet on Azure
* **Encryption**: CosmosDB always encrypts data at rest using encryption keys (‘service-managed keys’) that it also creates and manages. If extra security needed, use your own personal ‘customer-managed encryption key’ which you’re storing in azure key vault.

Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, application

Description automatically generated Graphical user interface, text, application, email

Description automatically generated Graphical user interface, text, email

Description automatically generated A picture containing diagram

Description automatically generated

Now click > Review+Create

* Goto cosmosdb resource > overview

Graphical user interface, text, application, email

Description automatically generated

* ‘Keys’ tab have primary and secondary keys which gives unlimited access to account. So never share those. Generally your application only use ‘primary master key’. Use ‘secondary master key’ only when you want to change primary key (rotating keys). So, temporarily use ‘secondary key’ while you change primary key. This is to make uninterrupted running application.

Graphical user interface, text, application, email

Description automatically generated

* URI end point+ Primary Key= Primary connection string
* URI endpoint+ secondary key=secondary connection string
* There are read-only version of these keys in ‘Read-only Keys’ tab for unlimited read access

**Creating a Container:**

* Go to ‘Data explorer’ tab
* Data explorer has full screen feature on top right corner

Graphical user interface, application, Word

Description automatically generated

* The above keys are temporary for security reasons. they do expire.
* Open full screen in read and write mode to focus only on data explorer

Graphical user interface, application

Description automatically generated

* Click ‘New Container’.
* First specify **new/existing database name** and specify **container name** to be created within that database
* In somecases you only have 1 container in DB. All the items in a container can be of different types and have different schemas, so you definitely not going to create 1 container for each type of entity in your data model.
* **Throughput**: performance level that you need the container to deliver.
  + higher number gives higher performance at higher cost
* **partitioning key** : partitioning has to do the way cosmosDB physically stores multiple documents together based on the partition key that you choose. Ie, common property in the data you writing to container .
* there are cases where you need multiple container in one DB, Because even though containers can provide unlimited storage, some data may require more or less throughput than others, and some data might have different partitioning needs than other data in order to deliver that unlimited storage.
* **Share throughput across containers** ::All containers in DB share throughput evenly. Uncheck this feature to control throughput at container level, since you’ll usually want to control throughput independently for each container in the database.
* Analytical store :: if on, Cosmos DB will automatically create a column store copy of your transactional data and keep it synchronized in near real time as you ingest data into your container. Then, using the Azure Synapse Link, you can use Azure Synapse Analytics for analytic workloads against the column store in the container with zero impact on the transactional workload.

Graphical user interface, application

Description automatically generated

* Now Create Container

**Creating Documents:**

* ‘Families DB’ has ‘Families container’. Now create ‘documents/Items’ inside this container (click new item inside ‘items’ tab and type json)
* ‘zipcode’ property is needed since it is partition key and every document also needs ‘id’ own property which is unique among all documents within partition key.
* So,Each item is unique in container by ‘zip code’ and ‘id’. In many cases omit the id (remove it from json) when creating new document/item and global unique id (GUID) generates automatically

{

    "familyName":"Smith",

    "address":{

        "addressLine":"1234 Main street",

        "city":"Chicago",

        "state":"IL",

        "zipCode":"60601"

    },

    "parents":[

        "peter",

        "Alice"

    ],

    "kids":[

        "Adam",

        "Jacqueline",

        "Joshua"

   ]

}

Type and save. Id generates automatically

Graphical user interface, text, application, email

Description automatically generated

* Note! Unlike relational databases , here
  + 1. schema not defined for this database because we don’t have enforced schema.
  + 2. single document represents what would typically be stored as multiple rows in separate relational tables joined by related keys. Eg: address would be a separate row in the address table joined to the family in a 1-1 relationship & parents and kids would be stored as separate rows in other tables joined to the family in a 1‑to‑many relationship.
  + from a data modeling perspective, get used to the **denormalized nature** of JSON documents when transitioning from the highly normalized world of relational databases.
  + with no enforced schema, it's easy to accommodate new family properties as they come up in the future. Eg: pets
  + instead of manual creation, these documents are generally created using code, or by migrating from external data sources
  + existing documents can be updated using ‘update’ button
* click ‘new sql query button’ for querying documents using sql

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

* the above query get 2 documents since both cities are ‘Chicago’ but both ‘zipcodes’ are different (ie, each document stored in different partition internally). So, this query is **‘Cross Partition Query’**
* SELECT \* FROM c WHERE c.address.zipcode=’60601’return only 1 document , the only family in that zipcode. Since you filtering on partition key, this is called **‘single partition key’**
* SELECT \* FROM c WHERE IS\_DEFINED(c.pets) >returns only 1 family (jones family) since only this has pets property >it’s a **‘Cross Partition Query’** since it is checking all zipcodes
* SELECT \* FROM c WHERE ARRAY\_LENGTH(c.kids) > 2 >>returns only 1 family (smith family)

**Management Options:**

1. Azure Portal
2. Azure CLI

Text

Description automatically generated

1. Powershell

Text

Description automatically generated

1. Notebooks

Graphical user interface, text, application

Description automatically generated

1. REST API (url or use .NET SDK wapper around REST API )

<https://docs.microsoft.com/en-us/rest/api/cosmos-db/create-a-database>

**Notebook:**

* Create DB, container

Graphical user interface, text, application, email

Description automatically generated

* %%upload magic :: Populate documents using json data from blob storage

Graphical user interface, text, application, email, Teams

Description automatically generated

* %%sql magic: to use sql to query json documents and show then as table rows

Graphical user interface, application

Description automatically generated

* dfsddfsd