Azure databases

Table of Contents

1. Azure Cosmos DB account

An Azure Cosmos DB account is an Azure resource that acts as an organizational entity for your databases.

> The account name (ID) can contain only lowercase letters, numbers, and the hyphen (-) character, and it must contain 3 to 31 characters.
>
> Because **documents.azure.com** is appended to the ID that you provide to create your URI, use a unique but identifiable ID.

1.1. Create an Azure Cosmos DB account

1.1.1. Azure Portal

1. Go to Azure Cosmos DB
2. Click create Azure Cosmos DB account
    - Subscription
    - Resource group
    - Account name (ID)
    - API
    - Location
    - Capacity node — e.g. provisioned throughput or serverless
    - Account type — e.g. non-production or production
    - Geo-redundancy — create a replicated version of your database in a second (paired) region
    - Multi-region writes — enables you to write to multiple regions at the same time
    - Availability zones

1.1.2. Azure CLI

1. Choose and store the name of the Azure Cosmos DB account in an environment variable to use later

    ```
    export NAME=[comos_db_account_name]
    ```

2. Create a new Azure Cosmos DB Account (the settings are displayed as a JSON object when finished)

```
az cosmosdb create \
  --name $NAME \
  --kind GlobalDocumentDB \
  --resource-group [group_name]
```

Example JSON object for an Azure Cosmos DB account

```
{
  "capabilities": [],
  "consistencyPolicy": {
    etc.
  },
  "databaseAccountOfferType": "Standard",
  "documentEndpoint": "https://cosmos123456.documents.azure.com:443/",
  "enableAutomaticFailover": false,
  "enableMultipleWriteLocations": false,
  "failoverPolicies": [
    etc.
  ],
  "id": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/learn-538b4606-e7ca-4205-9e28-
bdcdbce38302/providers/Microsoft.DocumentDB/databaseAccounts/cosmos123456",
  "ipRangeFilter": "",
  "isVirtualNetworkFilterEnabled": false,
  "keyVaultKeyUri": null,
  "kind": "GlobalDocumentDB",
  etc.
}
```

2. Azure Cosmos DB

Azure Cosmos DB is a globally distributed and elastically scalable database. It has a guaranteed low latency that is backed by a comprehensive set of Service Level Agreements (SLAs).

At the lowest level, Azure Cosmos DB stores data in atom-record-sequence (ARS) format. The data is then abstracted and projected as an API, which you specify when you are creating your database.

Azure Cosmos DB offers you five different consistency levels

- strong
- bounded staleness
- session
- consistent prefix
- eventual

All of the above is supported by a multi-model Azure Cosmos DB's approach, which provides you with the ability to use document, key-value, wide-column, or graph-based data.

2.1. Provisioning throughput

Adequate throughput is important to ensure you can handle the volume of transactions for your business needs.

In Azure Cosmos DB, you provision throughput for your containers to run writes, reads, updates, and deletes. You can provision throughput for an entire database and have it shared among containers within the database.

Throughput is reserved only for that container and it's evenly distributed among its physical partitions.

To scale throughput strategically, you need to estimate your throughput needs by estimating the number of operations you'll have to support at different times. If your requests consume all of the provisioned throughput, Azure Cosmos DB will rate-limit your requests. Operations will have to wait and retry, likely causing higher latency.

> If you attempt to use throughput higher than the one provisioned, your request will be rate-limited. When a request is rate-limited, the request has to be retried again after a specified interval.

2.2. Request unit

Azure Cosmos DB measures throughput using something called a request unit (RU). Request unit usage is measured per second, so the unit of measure is request units per second (RU/s). You must reserve the number of RU/s you want Azure Cosmos DB to provision in advance, so it can handle the load you've estimated, and you can scale your RU/s up or down at any time to meet current demand.

A single request unit, one RU, is equal to the approximate cost of performing a single GET request on a 1-KB document using a document's ID. Performing a GET by using a document's ID is an efficient means for retrieving a document, and thus the cost is small. Creating, replacing, or deleting the same item requires additional processing by the service, and therefore requires more request units.

The number of request units consumed for an operation changes depending on

- the document size,
- the number of properties in the document,
- the operation being performed,
- consistency,
- indexing policy, etc.

Multiply the number of consumed RUs of each operation by the estimated number of times each operation (write, read, update, and delete) will be executed per second. If you run several different queries on your data, you should understand how many RUs each query will consume. By summing the number of consumed RUs for each operation, you will be able to accurately estimate how many RUs to provision.

You provision the number of RUs on a per-second basis and you can change the value at any time in increments or decrements of 100 RUs. You’re billed on an hourly basis.

Request Unit considerations

Item size	As the size of an item increases, the number of RUs consumed to read or write the item also increases.
Item indexing	By default, each item is automatically indexed. Fewer RUs are consumed if you choose not to index some of your items in a container.
Item property count	Assuming the default indexing is on all properties, the number of RUs consumed to write an item increases as the item property count increases.
Indexed properties	An index policy on each container determines which properties are indexed by default. To reduce the RU consumption for write operations, limit the number of indexed properties.
Data consistency	The strong and bounded staleness consistency levels consume approximately two times more RUs on read operations when compared to that of other relaxed consistency levels.
Query patterns	The complexity of a query affects how many RUs are consumed for an operation. Factors that affect the cost of query operations include: • The number of query results • The number of predicates • The nature of the predicates • The number of user-defined functions • The size of the source data • The size of the result set • Projections
Script usage	As with queries, stored procedures and triggers consume RUs based on the complexity of their operations. As you develop your application, inspect the request charge header to better understand how much RU capacity each operation consumes.

Azure Cosmos DB guarantees that the same query on the same data always costs the same number of RUs on repeated executions.

When you create an account, you can provision a **minimum of 400 RU/s**, or a **maximum of 250,000 RU/s** in the portal.

2.3. Partition strategy

A partitioning strategy enables you to add more partitions to your database when need them. This scaling strategy is called **scale** out or **horizontal scaling**.

A partition key defines the partition strategy.

2.3.1. Partition key

A partition key (e.g. `userID` or `productID`) is the value by which Azure organizes your data into logical divisions.

It's set when you create a container and can't be changed. Selecting the right partition key is an important decision to make early in your development process.

It should aim to evenly distribute operations across the database to avoid hot partitions. A hot partition is a single partition that receives many more requests than the others, which can create a throughput bottleneck.

The amount of required RU's and storage determines the number of required physical partitions for the container, which are completely managed by Azure Cosmos DB. When additional physical partitions are needed, Cosmos DB automatically creates them by splitting existing ones. There is no downtime or performance impact for the application.

The storage space for the data associated with each **partition key can't exceed 20 GB**, which is the **size of one physical partition** in Azure Cosmos DB.

Best practices

- The more values your partition key has, the more scalability you have.
- To determine the best partition key for a read-heavy workload, review the top three to five queries you plan on using. The value most frequently included in the WHERE clause is a good candidate for the partition key.
- For write-heavy workloads, you'll need to understand the transactional needs of your workload, because the partition key is the scope of multi-document transactions.

2.3.2. Composite key

If your record is going to be **larger than 20 GB**, think about using a composite key instead so that each record is smaller. An example of a composite key would be `userID-date`, which would look like **CustomerName-08072018**. This composite key approach would enable you to create a new partition for each day a user visited the site.

2.4. Create a database and container

2.4.1. Azure Portal

1. Go to Azure Cosmos DB
2. Select an Azure Cosmos DB account
3. Click Data Explorer
4. Click New Container and specify the following settings
   - Database ID (e.g. Products)
   - Throughput (e.g. 1000 RU/s)
   - Container ID (e.g. Books)
   - Partition key (e.g. productId)
   - Default for remaining options
5. Create the new database and container (collection) by confirming with OK

2.4.2. Azure CLI

1. Create a new database in the account (the settings are displayed as a JSON object when finished)

   ```
   az cosmosdb sql database create \
     --account-name $NAME \
     --name "[database_name]" \
     --resource-group [group_name]
   ```

   Example JSON object for a database

   ```
   {
     "id": "/subscriptions/…/cosmos123456/sqlDatabases/Products",

     "location": null,
     "name": "Products",
     "resource": {
       etc.
     },
     "resourceGroup": "learn-538b4606-e7ca-4205-9e28-bdcdbce38302",
     "tags": null,
     "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases"
   }
   ```

2. Create a collection (container) with the specified partition key and throughput values

```
az cosmosdb sql container create \
  --account-name $NAME \
  --database-name "[database_name]" \
  --name "[collection_name]" \
  # example location is Germany West Central
  --location "Germany West Central"
  --partition-key-path "/[path]" \
  # example throughput is 1000
  --throughput 1000 \
  --resource-group [group_name]
```

Example JSON object for a collection

```json
{
  "id":
"/subscriptions/…/cosmos123456/sqlDatabases/Products/containers/Clothing",
  "location": null,
  "name": "Clothing",
  "resource": {
    etc.
  },
  "resourceGroup": "learn-538b4606-e7ca-4205-9e28-bdcdbce38302",
  "tags": null,
  "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers"
}
```

3. Azure Cosmos DB APIs

Azure Cosmos DB provides five APIs

1. SQL (relational database)
2. Gremlin (graph database)
3. MongoDB (document database)
4. Azure Table (currently requires a separate account)
5. Cassandra (currently requires a separate account)

Decision process

- existing database → use current API to reduce migration tasks
- emerging / changing schema → use document database, e.g. Core (SQL)
- relationships between items → use graph databases to store metadata
- key-value pairs → Core (SQL) API offers better querying with improved indexing than Table API

Table 1. Decision criteria matrix

	Core (SQL)	MongoDB	Cassandra	Azure Table	Gremlin
New projects being created from scratch	yes				
Existing MongoDB, Cassandra, Azure Table, or Gremlin data		yes	yes	yes	yes
Analysis of the relationships between data					yes
All other scenarios	yes				

3.1. Core (SQL) API

Core (SQL) is the default API for Azure Cosmos DB, which provides you with a view of your data that **resembles a traditional NoSQL document store**. You can query the hierarchical JSON documents with a SQL-like language. Core (SQL) uses JavaScript's type system, expression evaluation, and function invocation.

3.1.1. Querying

Core (SQL) provides several familiar SQL statements and clauses

- SELECT
- FROM
- WHERE
- BETWEEN
- COUNT
- SUM
- MIN
- MAX
- ORDER BY

Example query

```
SELECT c.productName FROM Items c
```

3.1.2. Use cases

Recommended for e-commerce, product catalogs, etc.

Decision criteria

- searchable
- filter and sort data based on different categories (SQL queries)
- region supported languages
- semi*structured data / schemaless data store
- flexible and scalable schema (unknown data)
- quickly add new categories
- low downtime

3.2. MongoDB API

Azure Cosmos DB's API for MongoDB supports the MongoDB wire protocol. his API allows existing MongoDB client SDKs, drivers, and tools to interact with the data, as if they are running against an actual MongoDB database.

The data is stored in document format, which is the same as using Core (SQL). Azure Cosmos DB's API for MongoDB is currently compatible with 3.2 version of the MongoDB wire protocol.

3.2.1. Querying

Example query

```
db.Items.find({},{productName:1,_id:0})
```

3.2.2. Use cases

Recommended for historical order data

Decision criteria

- data in different formats
- semi-structured data
- low downtime regarding data migration (import and reuse MongoDB database)
- reuse existing code such as MongoDB queries (`mongodump` and `mongorestore`)

3.3. Cassandra API

Azure Cosmos DB's support for the Cassandra API makes it possible to query data by using the Cassandra Query Language (CQL), and your data will appear to be a partitioned row store.

Cosmos DB's Cassandra API currently supports version 4 of the CQL wire protocol.

3.3.1. Querying

Azure Cosmos DB provides several familiar CQL statements and clauses

- CREATE KEYSPACE
- CREATE TABLE
- ALTER TABLE
- USE
- INSERT
- SELECT
- UPDATE
- BATCH (Only unlogged commands are supported)
- DELETE

Example query

```
-- create table that stores JSON info
CREATE TABLE Catalog.Items(id text, productName text, description text, supplier text,
quantity int, unitCost float, retailPrice float, categories map<text,text>, primary
key (id));

-- retrieve product name
SELECT id, productName FROM catalog.items
```

3.3.2. Use cases

Recommended for web analytics, chat features

Decision criteria

- experience with Cassandra Query Language (CQL)
- app based on Cassandra
- fixed schema
- reuse existing code with minimal changes

3.4. Azure Table API

Azure Cosmos DB's Azure Table API provides support for applications that are written for Azure Table Storage that need premium capabilities like global distribution, high availability, scalable throughput. The original Table API only allows for indexing on the Partition and Row keys; there are no secondary indexes. Storing table data in Cosmos DB automatically indexes all the properties, and requires no index management.

Table Storage is charged on the size of data rather than how often it is accessed.

3.4.1. Querying

Querying is accomplished by using OData and LINQ queries in code, and the original REST API for GET operations.

Example query

```
SELECT i.productName FROM Items i
```

3.4.2. Use cases

Recommended for storing IoT data

Decision criteria

- seldom update of data
- key-value pairs
- migrating a legacy Azure Table Storage database

3.5. Gremlin (graph) API

Choosing Gremlin as the API provides a graph-based view over the data. A graph-based view on the database means data is either a vertex (which is an individual item in the database), or an edge (which is a relationship between items in the database).

You typically use a traversal language to query a graph database, and Azure Cosmos DB supports Apache Tinkerpop's Gremlin language.

This kind of graph might be useful when you are creating a product recommendation application.

3.5.1. Querying

For example queries see Identify the technology options.

3.5.2. Use cases

Recommended for product recommendations, tracking services

Decision criteria

- rank products
- assign weight values to the relationships between products
- store relationship counter as metadata

4. References

4.1. MS modules

- Create an Azure Cosmos DB database built to scale
- Choose the appropriate API for Azure Cosmos DB === MS docs

4.2. MS docs

4.3. Free learning platform

- Azure Portal sandbox (time limit