



# Azure Cosmos DB Monitoring and Optimization

Emmanuel Deletang

[Emmanuel.Deletang@Microsoft.com](mailto:Emmanuel.Deletang@Microsoft.com)

TSP GBB nosql EMEA

# AGENDA

Reviewing Azure Cosmos DB

Monitor your cosmodb

Troubleshooting

Tips and tricks

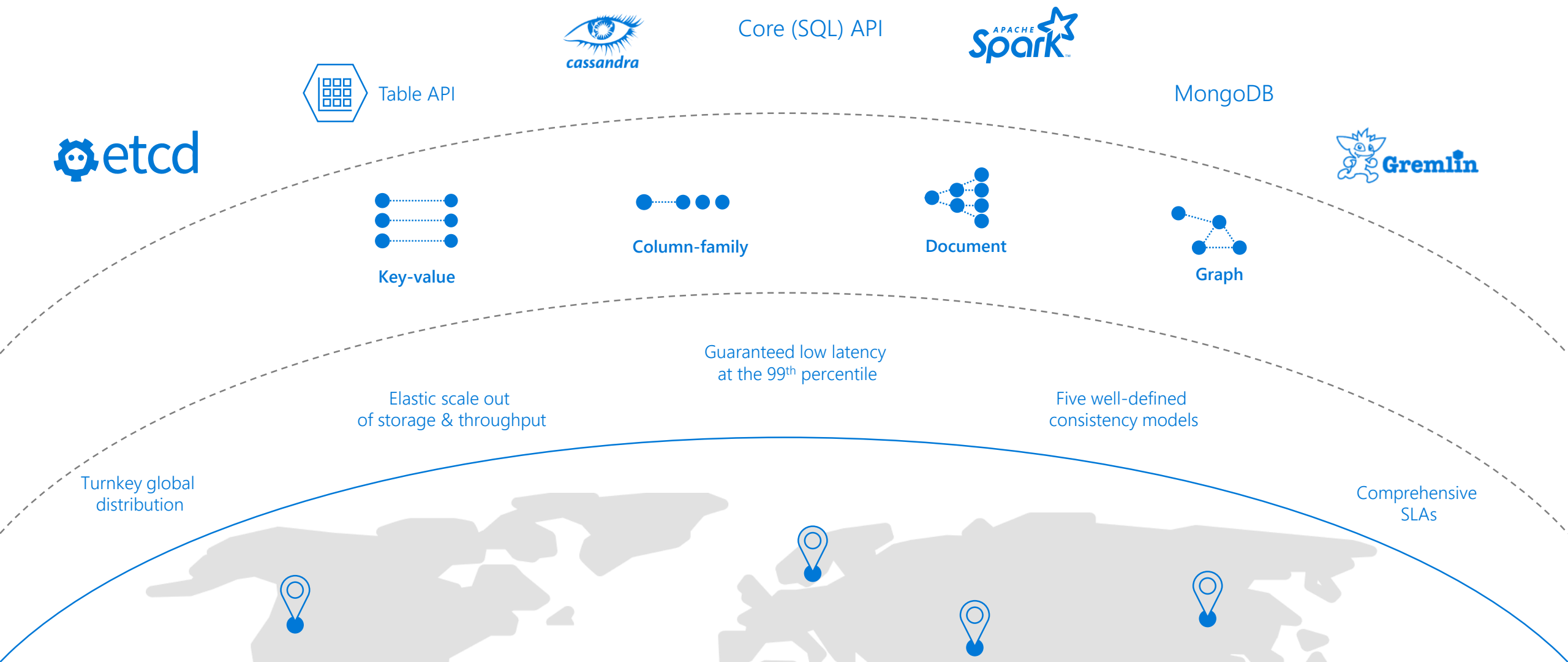
- Modeling & Planning tips and tricks

- Partitioning mandatory

- Querying be careful

- Programming best practice

# AZURE COSMOS DB



# AZURE COSMOS DB

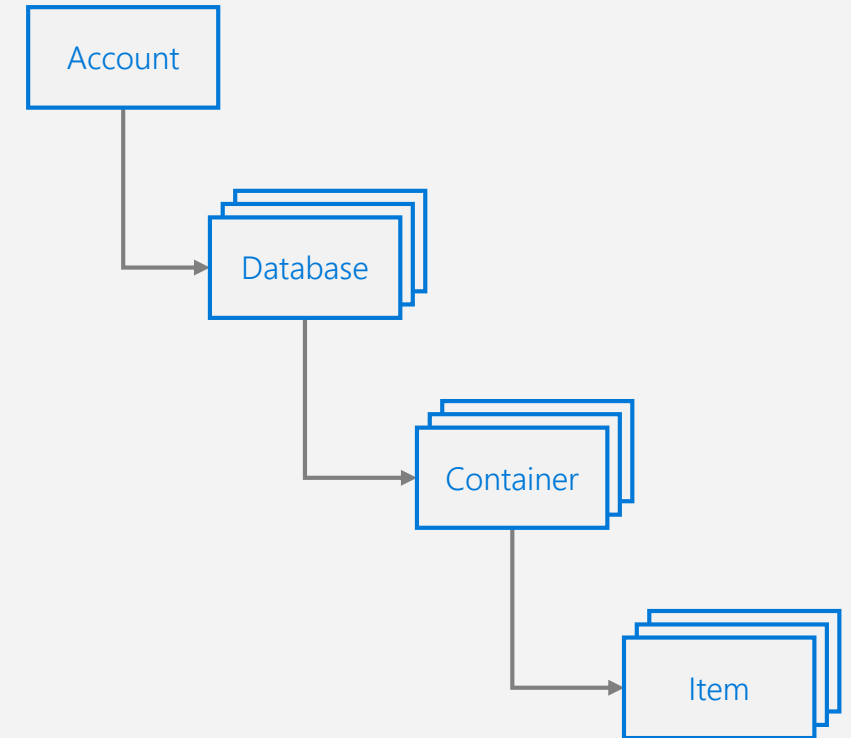
A FULLY-MANAGED GLOBALLY DISTRIBUTED DATABASE SERVICE BUILT TO GUARANTEE  
EXTREMELY LOW LATENCY AND MASSIVE SCALE FOR MODERN APPS



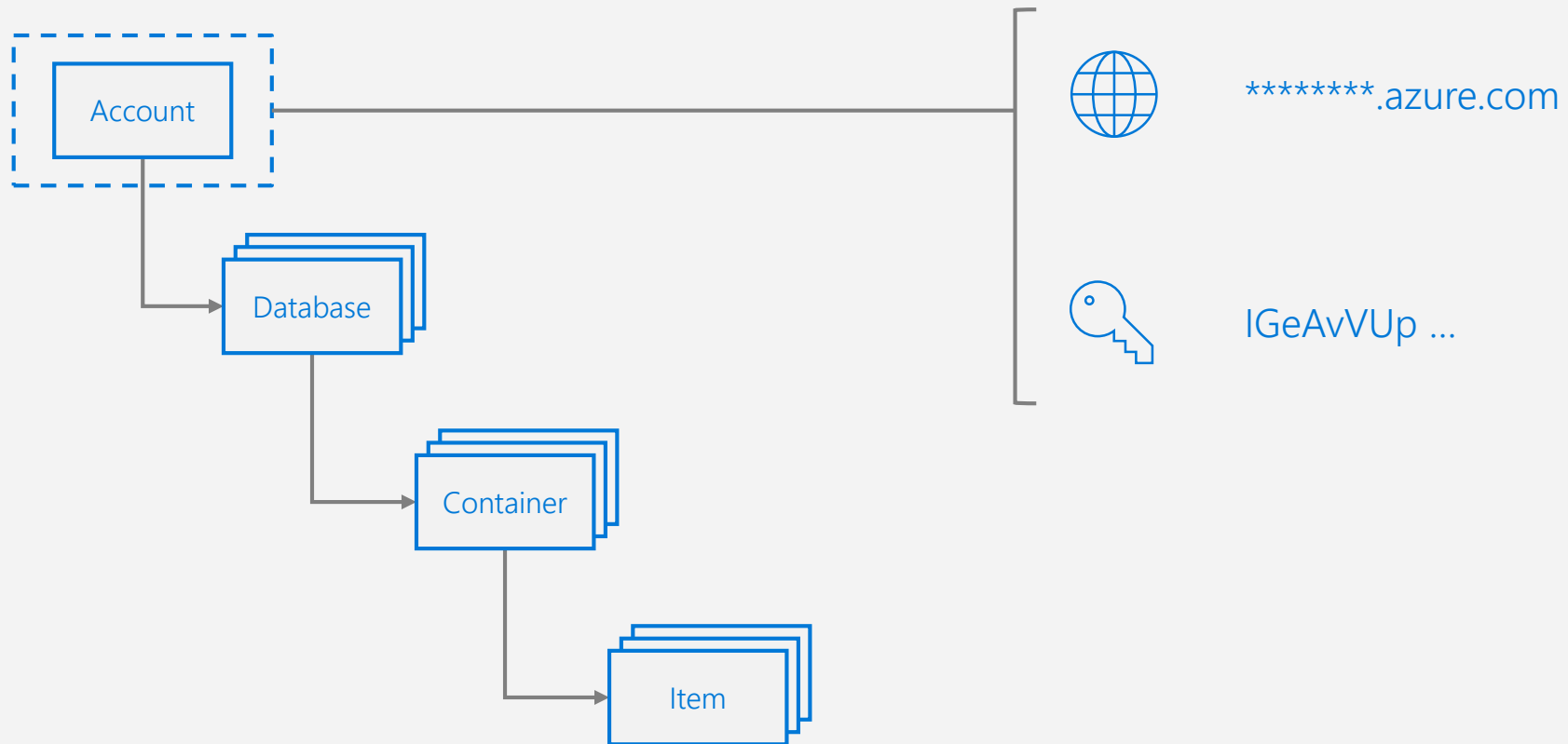
# RESOURCE MODEL

Leveraging Azure Cosmos DB to automatically scale your data across the globe

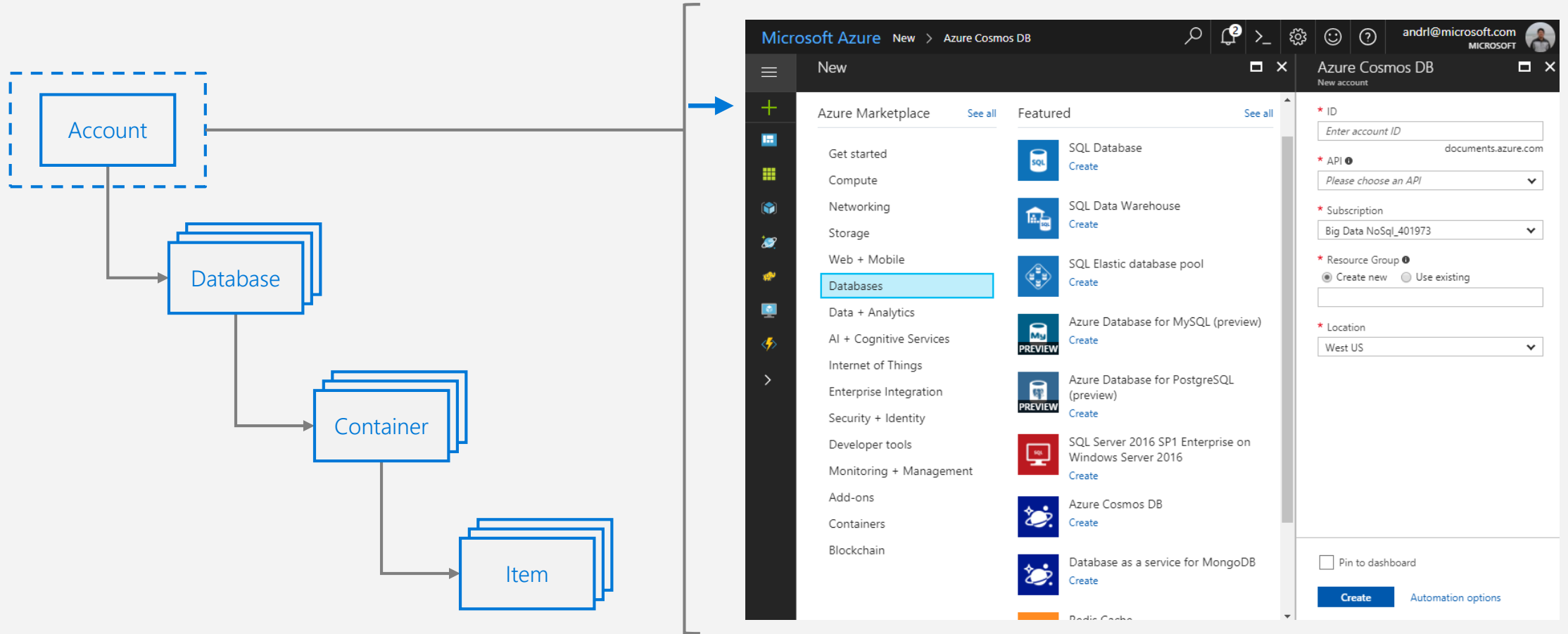
*This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.*



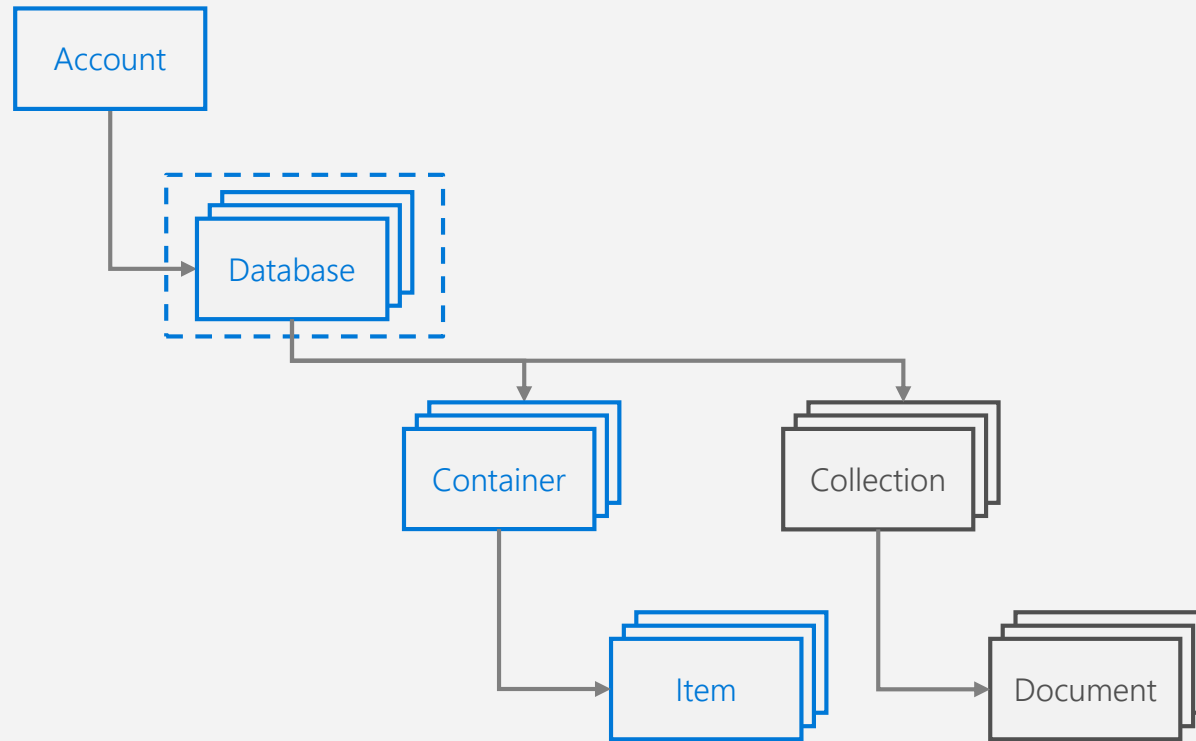
# ACCOUNT URI AND CREDENTIALS



# CREATING ACCOUNT

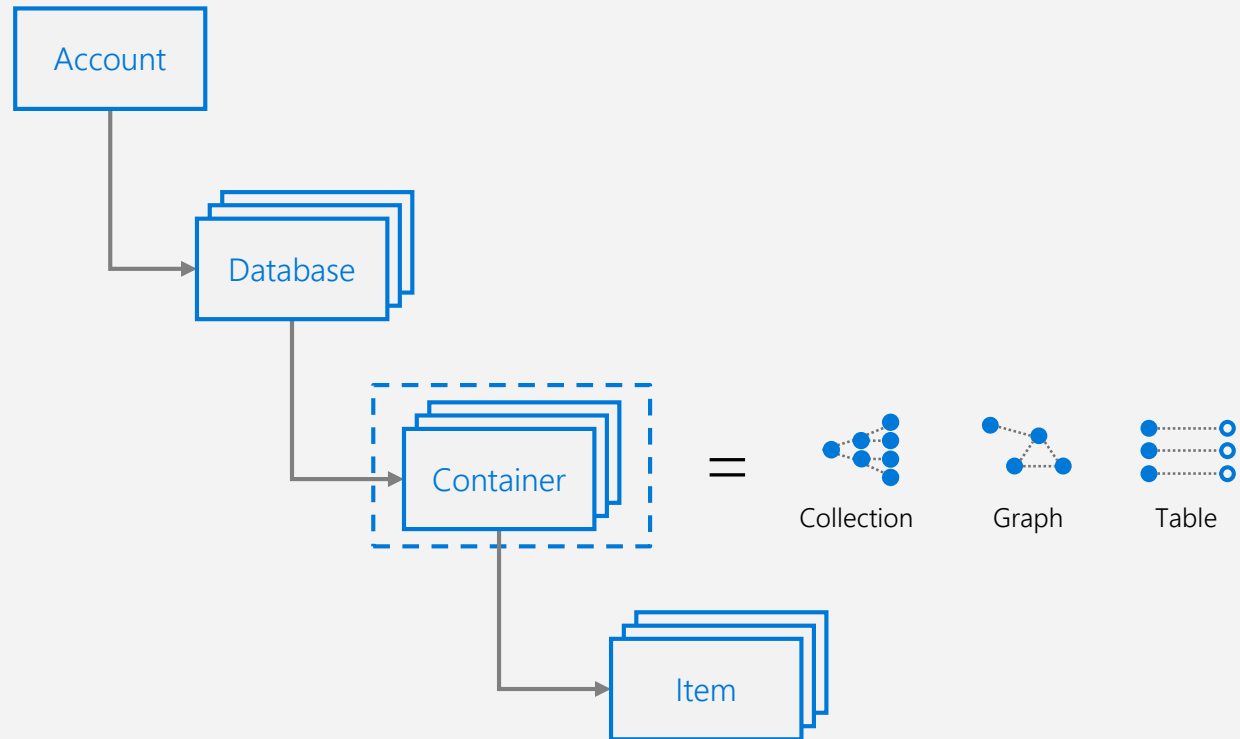


# DATABASE REPRESENTATIONS

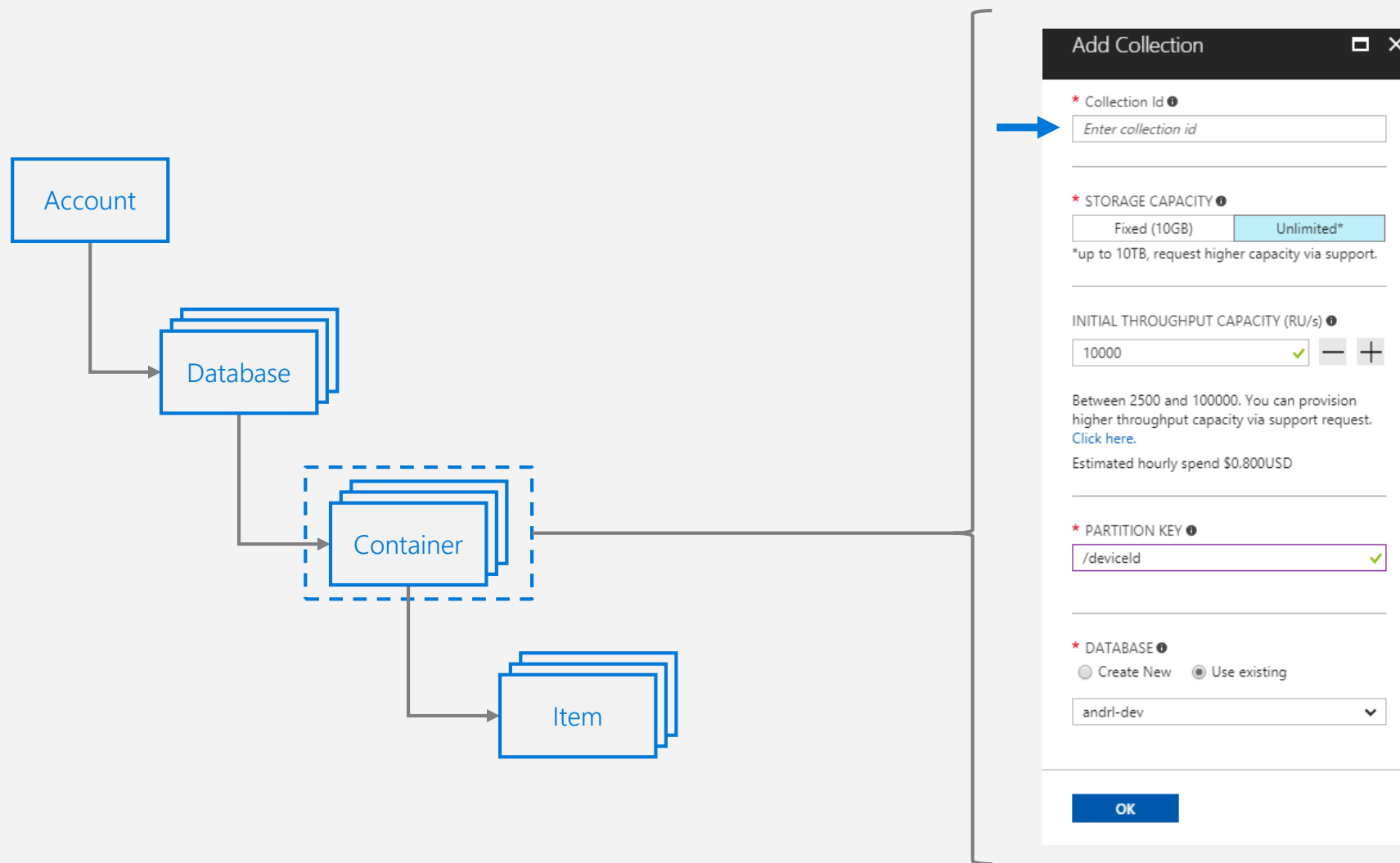




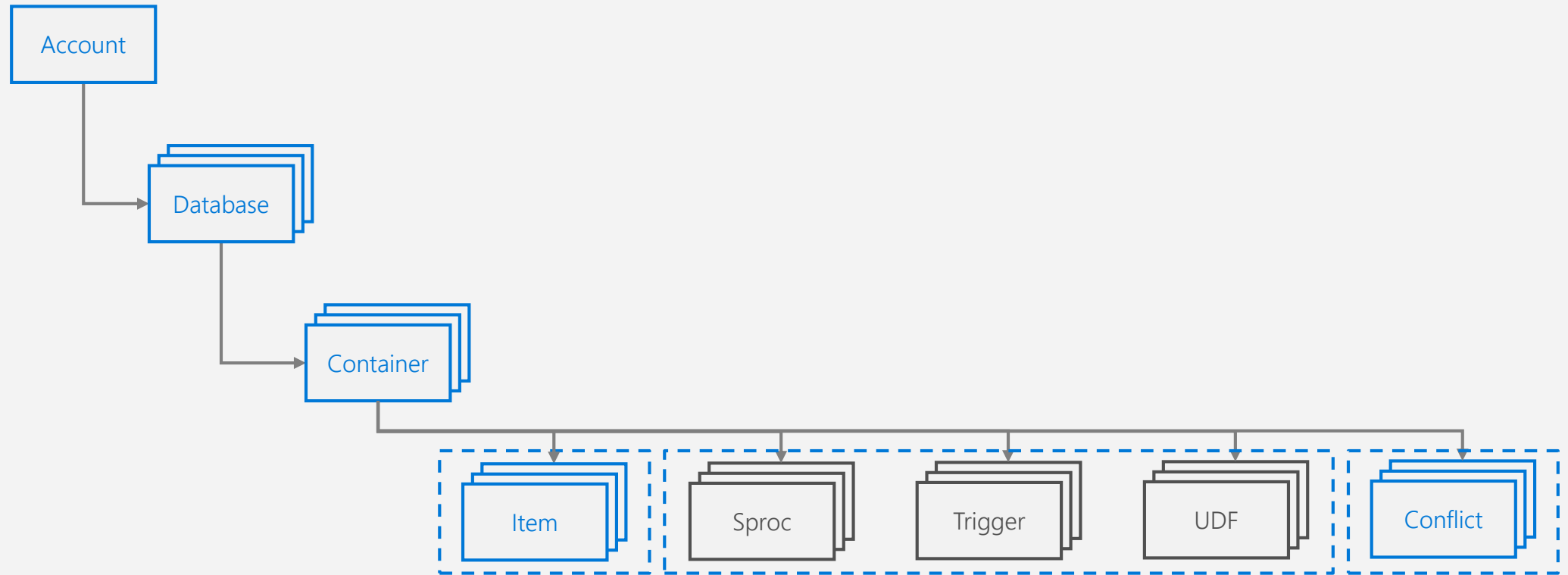
# CONTAINER REPRESENTATIONS



# CREATING COLLECTIONS – SQL API



# CONTAINER-LEVEL RESOURCES



## QUESTION

What happen to my car if I made nothing  
no fuel  
No maintenance  
No ....



AZURE COSMOS DB

**MANAGED** doesn' t mean no monitoring



# MONITORING AND OPTIMIZE COSMOSDB

What you can do

- USE AZURE MONITOR

- CHECK LOGS

- USE Azure ALERTS

- USE Azure ADVISORS

Setup to DO:

- ENABLE MONITORING

- ENABLE LOGGING

- SETUP ALERT (NO ALERT BY DEFAULT)

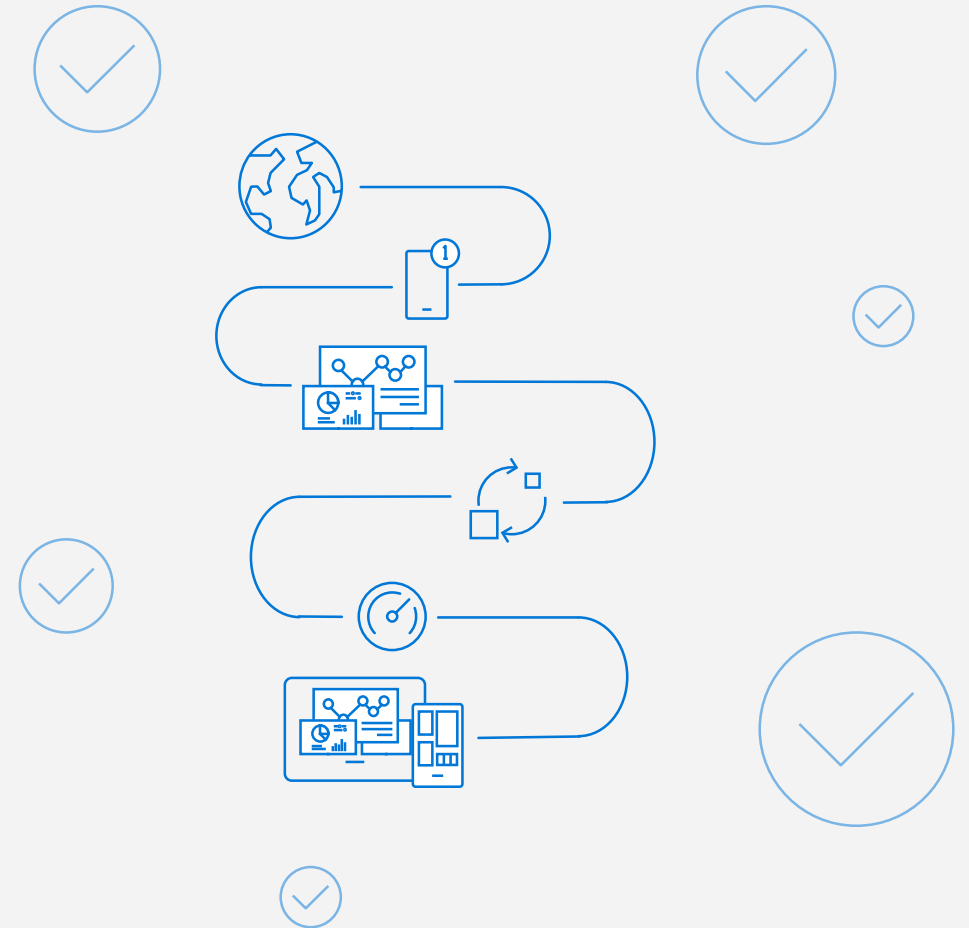
# MONITORING

Help to have an overview

Check 80 % of your works in 20 % of your time

Monitor the performance

Monitor the failures



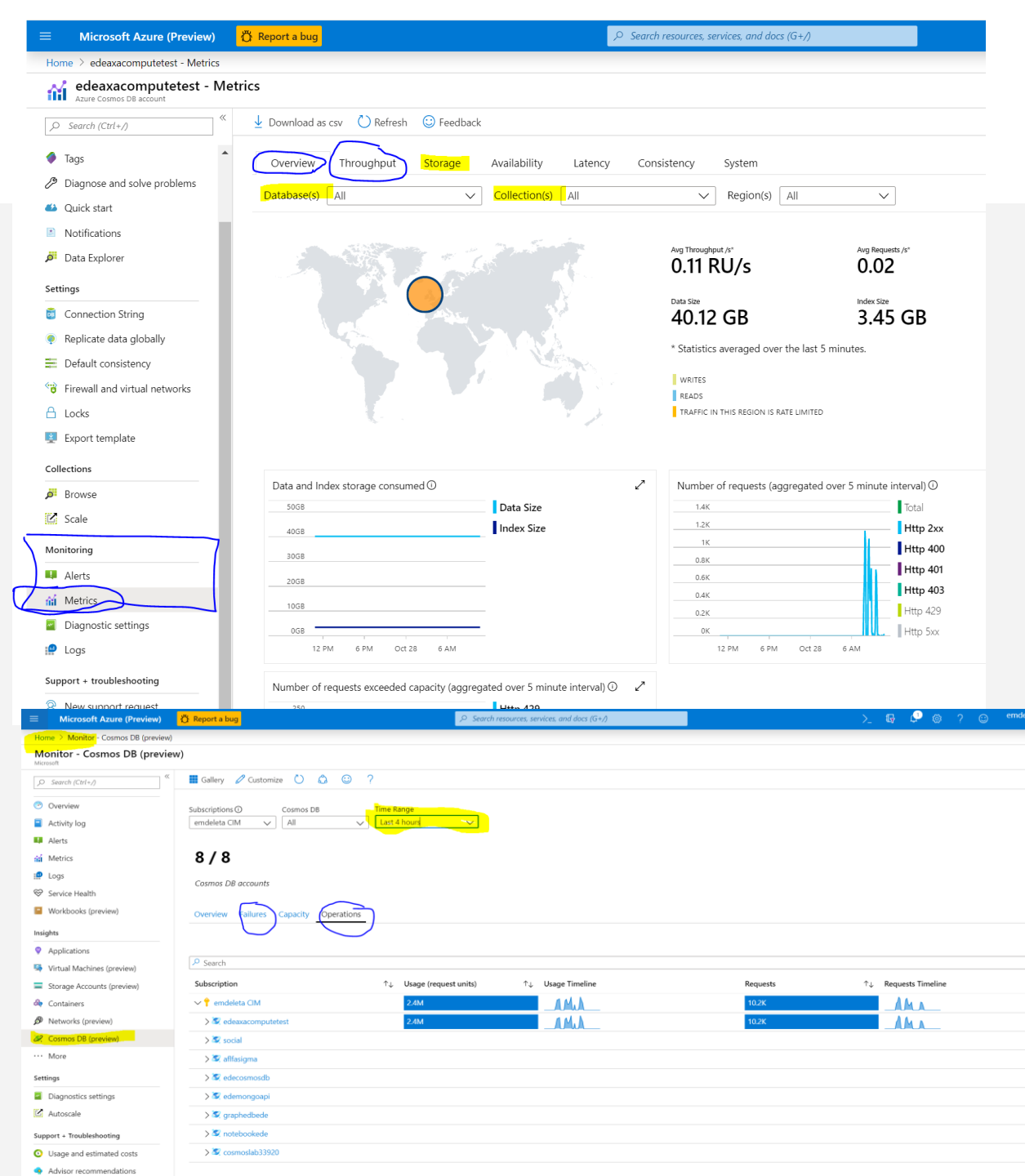
# MONITOR COSMOSDB

## Use though Cosmosdb portal Monitoring

Use azure monitor though portal to check the overall performance  
Check throughput and storage part  
to check the performance and 429 error  
To check the partitioning

## Use though azure Monitor

Global View of all your DB  
View of operations  
view of capacity  
View of Failures





# LOGGING

## WHAT IS LOGGED BY AZURE DIAGNOSTIC LOGS

All authenticated backend requests across all protocols and APIs

- Includes failed requests

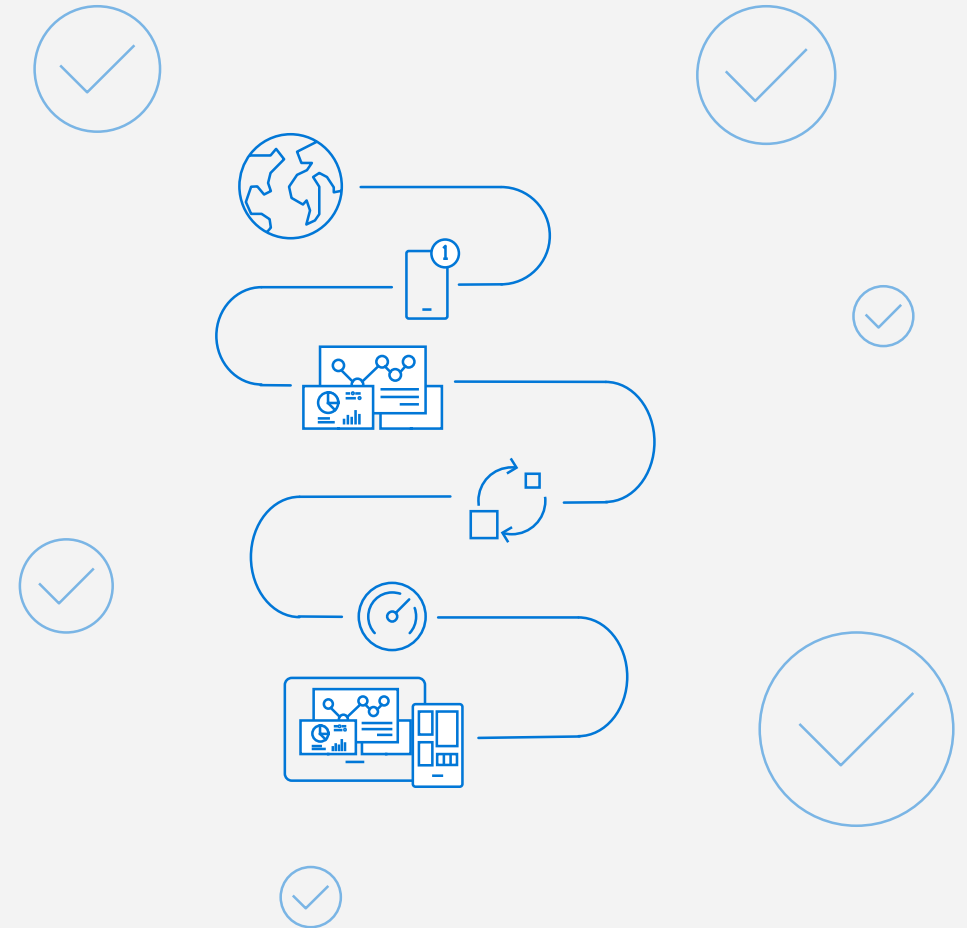
Database operations

- Includes CRUD operations on all resources

Account Key operations

Unauthenticated requests

- Requests that result in a 401 response



# VIEWING LOGS IN LOG ANALYTICS

## ENABLE LOGGING

Diagnostic Logs for Azure Services are opt-in. You should first enable logging (using the Portal, CLI or PowerShell).

Logs take, on average, about two hours to be made available.

## LOG ANALYTICS

If you selected the **Send to Log Analytics** option when you turned on diagnostic logging, diagnostic data from your collection is forwarded to Log Analytics.

From within the Log Analytics portal experience, you can:

- Search logs using the expanded Log Analytics query language
- Visualize the results of a query as a graph
- Pin a graph visualization of a query to your Azure portal dashboard

# THE MAGICS REQUEST

All the event in Cosmosdb by categories

```
AzureDiagnostics | where ResourceProvider=="MICROSOFT.DOCUMENTDB" | summarize count () by Category
```

Add the type

```
category == «»
```

```
AzureDiagnostics | where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="DataPlaneRequests" | take 10
```

Order or sort by

```
duration_s and requestCharge_s ( time and cost )
```

```
AzureDiagnostics | where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category == "MongoRequests" | sort by  
todouble(requestCharge_s) desc
```

```
AzureDiagnostics | where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category == "MongoRequests" | sort by  
todouble(duration_s) desc
```

filter on your data ( database )

To find the costly request

```
AzureDiagnostics | where todouble(requestCharge_s) > 1 and databaseName_s == "RCS" | sort by todouble(requestCharge_s) desc
```

To show information

```
AzureDiagnostics | where ResourceProvider=="MICROSOFT.DOCUMENTDB" and Category=="MongoRequests" | project TimeGenerated ,  
todouble(duration_s) | render timechart
```

DEMO

# Viewing Logs in Log Analytics

# ALERTING

## Setup alert to automate administrative task :

- Alert on change
- Alert on most popular error :
  - 429 Too Many Request
  - 413 Entity Too Large
  - 400 Bad Request ..... <https://docs.microsoft.com/en-us/rest/api/cosmos-db/http-status-codes-for-cosmosdb>
- Add any action
  - Email
  - Azure fonction can be add ( all you want by code ) (autoscale ? )

# ADVISORS

Azure Advisors can help you to check on Cosmosdb

[Feedback](#) [Download as CSV](#) [Download as PDF](#)


Your recommendations have been loaded


**Subscriptions:** 4 of 6 selected – Don't see a subscription? [Open Directory](#) + [Subscription settings](#)

4 subscriptions


Azure Cosmos DB accounts


Active

 **High Availability**





You are following all of our high availability recommendations  
[See list of high availability recommendations](#)

 **Security**





You are following all of our security recommendations  
[See list of security recommendations](#)

 **Performance**




You are following all of our performance recommendations  
[See list of performance recommendations](#)

 **Operational Excellence**



You are following all of our operational excellence recommendations  
[See list of operational excellence recommendations](#)

 **Cost**

954 USD savings/yr \*

**2 Recommendations**

0 High impact

0 Medium impact

2 Low impact

DEMO

# Advisors

# TIPS AND TRICKS

- Enable LOGS
  - Check the performance with azure monitor
  - Configure Alert to automate task
  - Use advisors ( free and can be helpful)
- 
- When you have done this , you can begin to think optimization
- 
- To AVOID the worst call of your life ?
    - Put a lock on delete for production environnement 😊



AZURE COSMOS DB

Optimize your CosmosDB



# OPTIMIZE

- Optimization is not a one-shot task
- Optimization depend on the need
  - Performance
  - Cost
  - Need to find a balance
  - Not think like RDBMS DATABASE
  - Not only DBA should made optimization
- A Database is a Database ( cardinality , no magic)

AZURE COSMOS DB

Some mandatory concept



What are Request Units (RUs)?

# WHAT ARE REQUEST UNITS (RUS)?

- In Cosmos DB, expected performance must be **provisioned**
- Expressed in **Request Units per second (RU/s)**
  - Represents the "cost" of a request in terms of CPU, memory and I/O
- Performance can be provisioned:
  - at the database-level
  - at the collection-level
  - or both
- Provisioned performance **can be changed programmatically** with API calls

```
var offer = client.CreateOfferQuery()  
    .Where(o => o.ResourceLink == collection.SelfLink)  
    .AsEnumerable()  
    .Single();  
await client.ReplaceOfferAsync(new OfferV2(offer, 10000));
```

# WHAT ARE REQUEST UNITS (RUS)?

## Each request consumes # of RU

1 RU = 1 read of 1 KB document

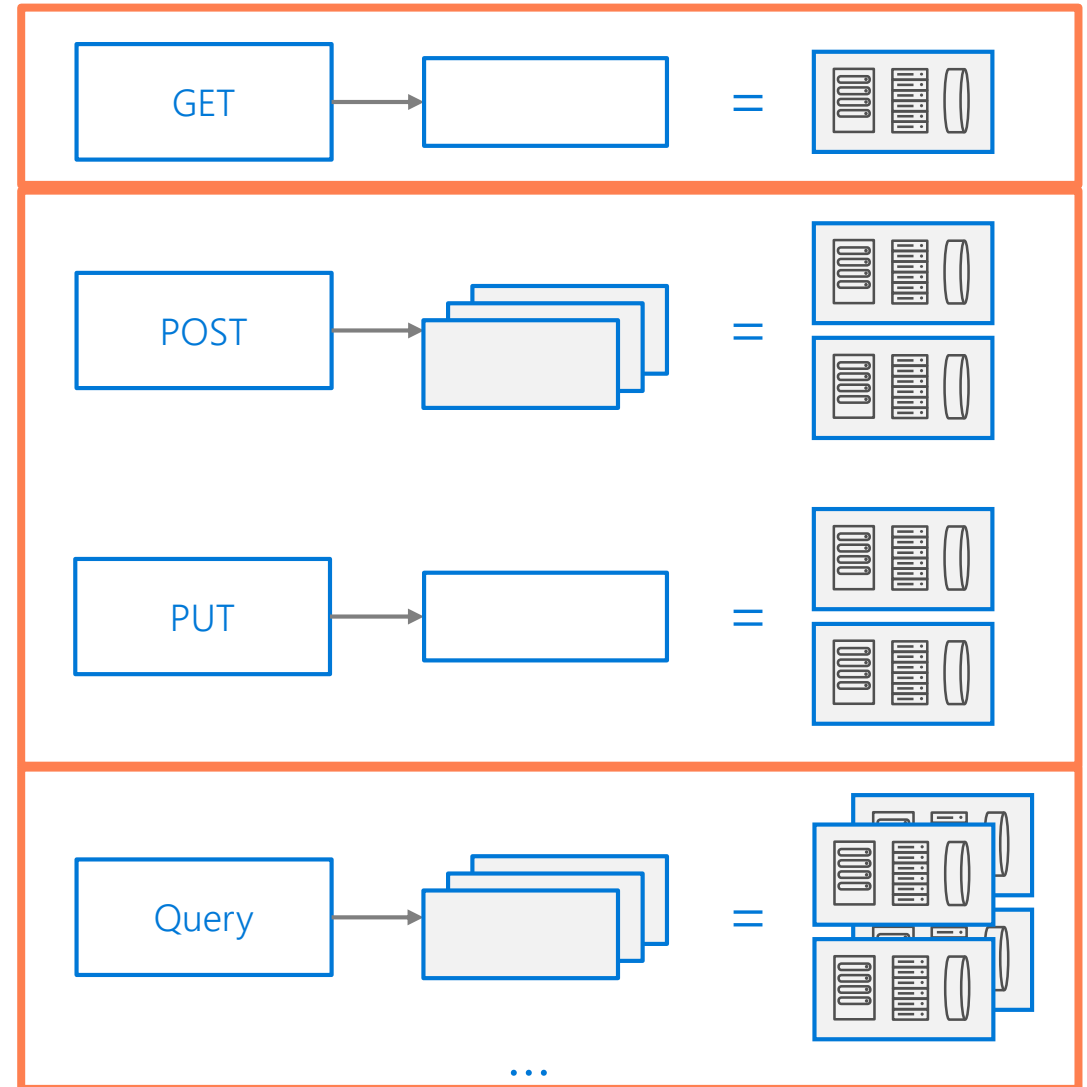
10 RU = 1 read of 100 KB document

5 RU = 1 write of a 1KB document

50 RU = 1 write of a 100KB document

Query: Depends on query & documents involved

(Indexing also affects RU cost)



# Database Level Throughput

You can provision throughput at the database level instead of individually for each container

Throughput is *shared* among each designated container within the database

You can provision dedicated throughput for some containers

# DATABASE VS CONTAINER LEVEL THROUGHPUT

In general, **container level throughput** is a good choice. This leads to predictable performance since each container is guaranteed its provisioned RU's

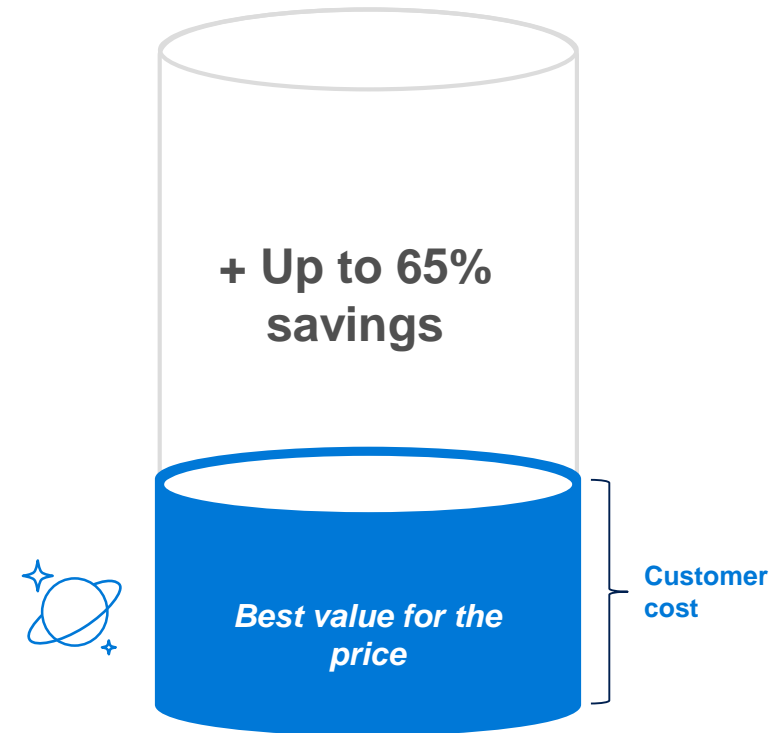
Choosing Database Level Throughput can also be a good option if:

- You are migrating many containers in a lift and shift migration (from Table Storage, MongoDB, or Cassandra) and does not know how much throughput to set for each one
- You have containers that are timeshared
- Multitenant applications where each user is represented by a separate container



# COSMOS DB RESERVED CAPACITY CAN PROVIDE UP TO 65% SAVINGS

## Save up to 65% with Azure Cosmos DB reserved capacity pricing



*Best elasticity for the price*

*No need to segment  
into read and write workloads –  
w/ unified and normalized  
throughput currency - RUs*

*Saturation of provisioned  
capacity via sub-core  
multiplexing*

*Total Time To Live (TTL) is  
free*

*Deep integration w/ Azure  
Networking*

*Enterprise-ready (security,  
compliance, encryption) at  
no additional cost*



Why is this query consuming  
a lot of RUs?

# QUERY TUNING

## Some important Cosmos DB query performance factors include:

### Provisioned throughput

Measure RU per query, and ensure that you have the required provisioned throughput for your queries

### Partitioning and partition keys

Favor queries with the partition key value in the filter clause for low latency

### SDK and query options

Follow SDK best practices like direct connectivity, and tune client-side query execution options

### Network latency

Account for network overhead in measurement, and use multi-homing APIs to read from the nearest region

# QUERY TUNING

**Additional important Cosmos DB query performance factors include:**

Indexing Policy

Ensure that you have the required indexing paths/policy for the query

Query Complexity

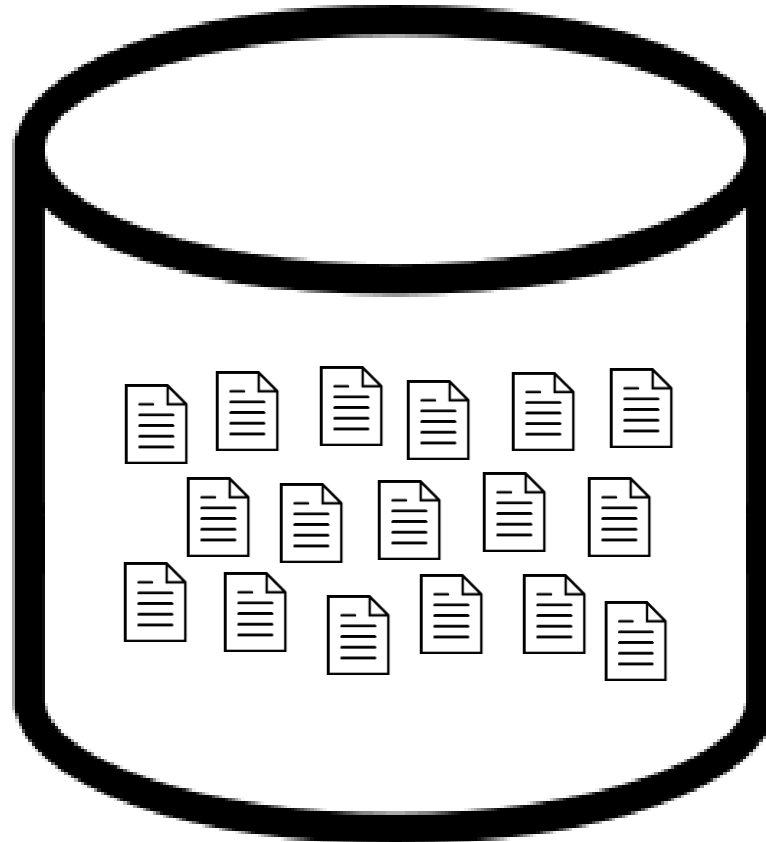
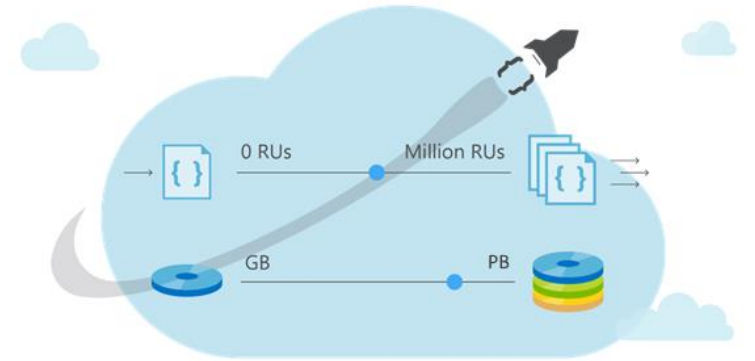
Use simple queries to enable greater scale.

Query execution metrics

Analyze the query execution metrics to identify potential rewrites of query and data shapes

# WHAT IS PARTITIONING?

- Splitting documents over **multiple servers**
- Enables **horizontal scaling**

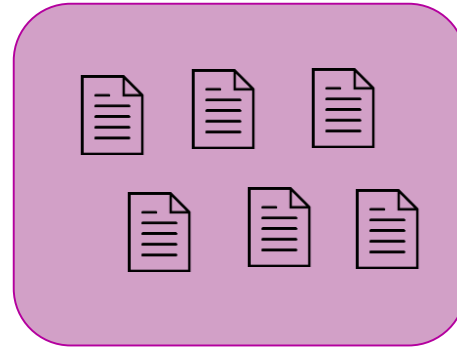


# WHAT IS PARTITIONING?

- Within a collection, documents are split into groups called **logical partitions**



Logical partition



Logical partition



Logical partition

# WHAT IS PARTITIONING?

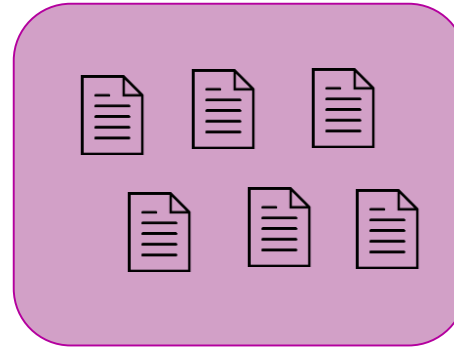
- These logical partitions group documents according to the collection's partition key

`'userId': 'Serge'`



Logical partition

`'userId': 'thomas'`



Logical partition

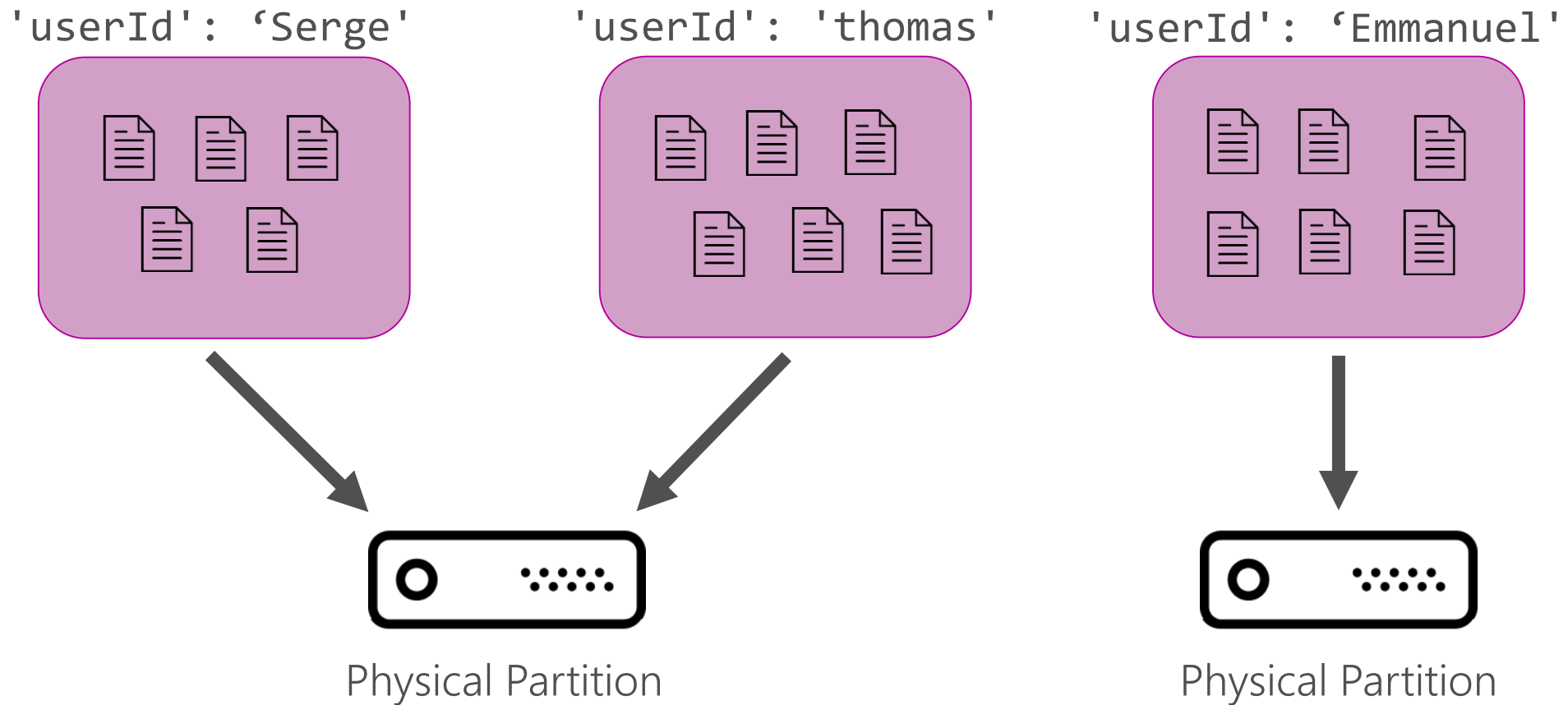
`'userId': 'Emmanuel'`



Logical partition

# WHAT IS PARTITIONING?

- Cosmos DB seamlessly shuffles logical partitions across physical servers to scale horizontally





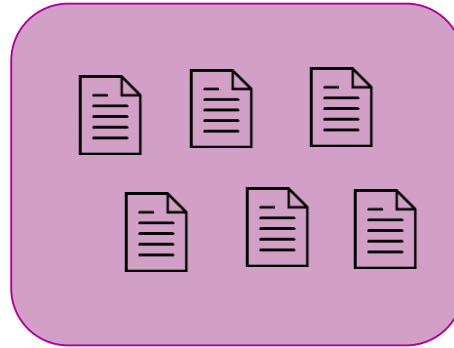
# WHAT IS PARTITIONING?

- A good partition key ensures **well-balanced partitions**
  - Both in terms of storage and throughput
- Ideally, read queries should get all their results from one partition

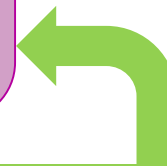
'userId': 'Serge'



'userId': 'thomas'



'userId': 'Emmanuel'



```
SELECT * FROM u WHERE u.userId = 'thomas'
```

# PARTITIONS

## Best Practices: Design Goals for Choosing a Good Partition Key

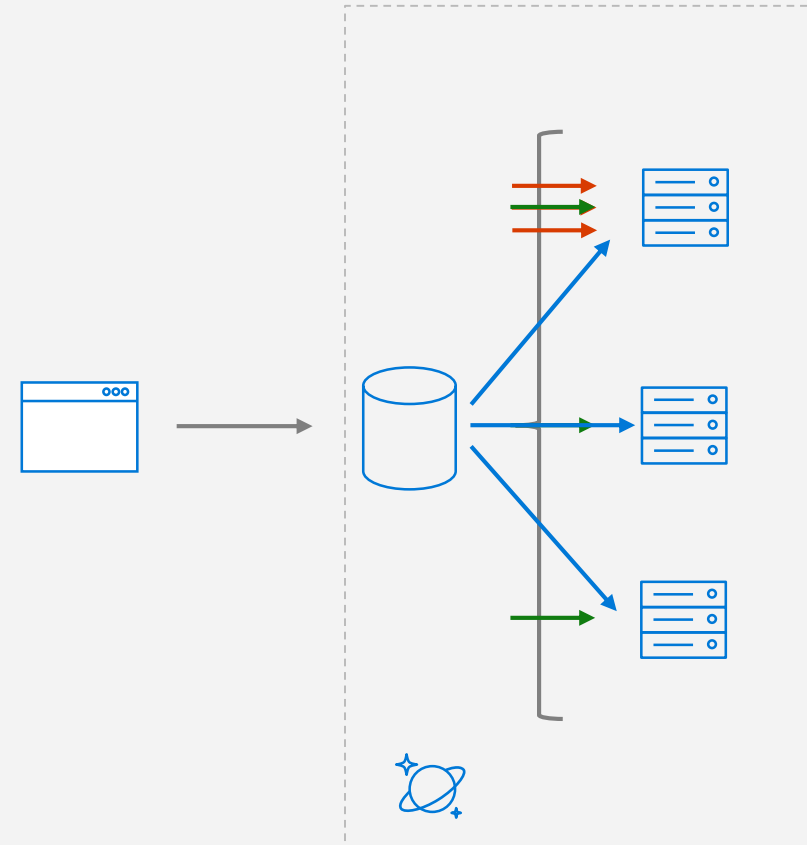
- Distribute the overall request + storage volume
  - Avoid "hot" partition keys
- Partition Key is scope for multi-record transactions and routing queries
  - Queries can be intelligently routed via partition key
  - Omitting partition key on query requires fan-out

## Steps for Success

- Ballpark scale needs (size/throughput)
- Understand the workload
- # of reads/sec vs writes per sec
  - Use pareto principal (80/20 rule) to help optimize bulk of workload
  - For reads – understand top 3-5 queries (look for common filters)
  - For writes – understand transactional needs

## General Tips

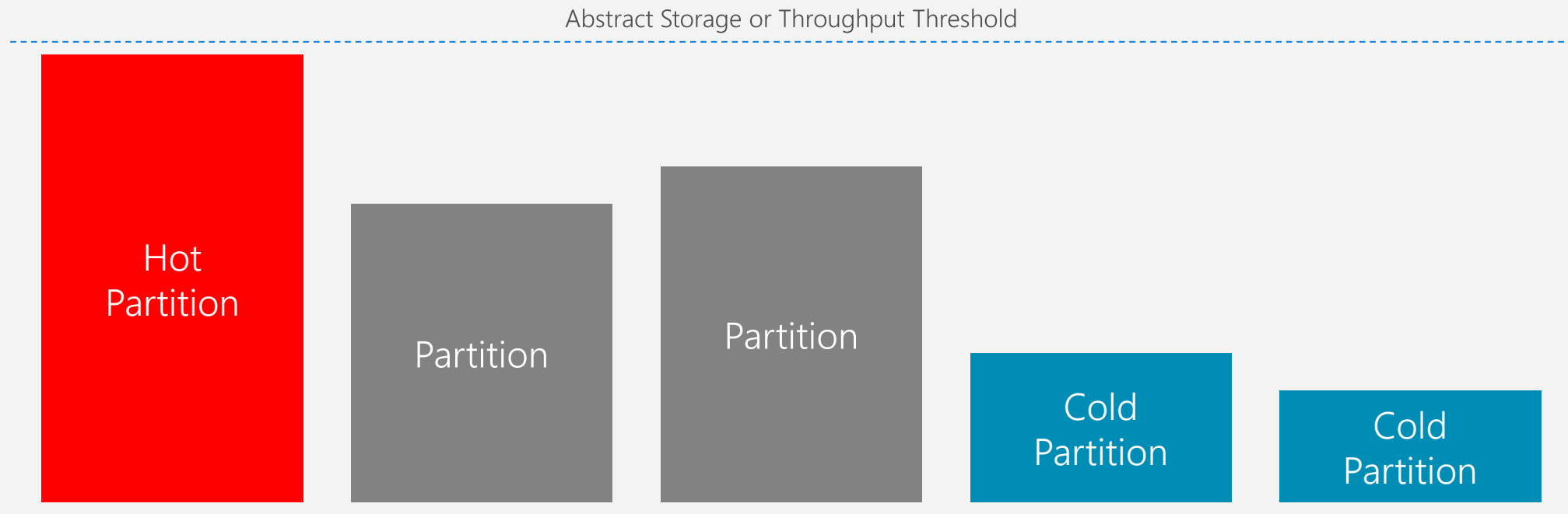
- Build a POC to strengthen your understanding of the workload and iterate (avoid analyses paralysis)
- Don't be afraid of having too many partition keys
  - Partitions keys are logical
  - More partition keys → more scalability



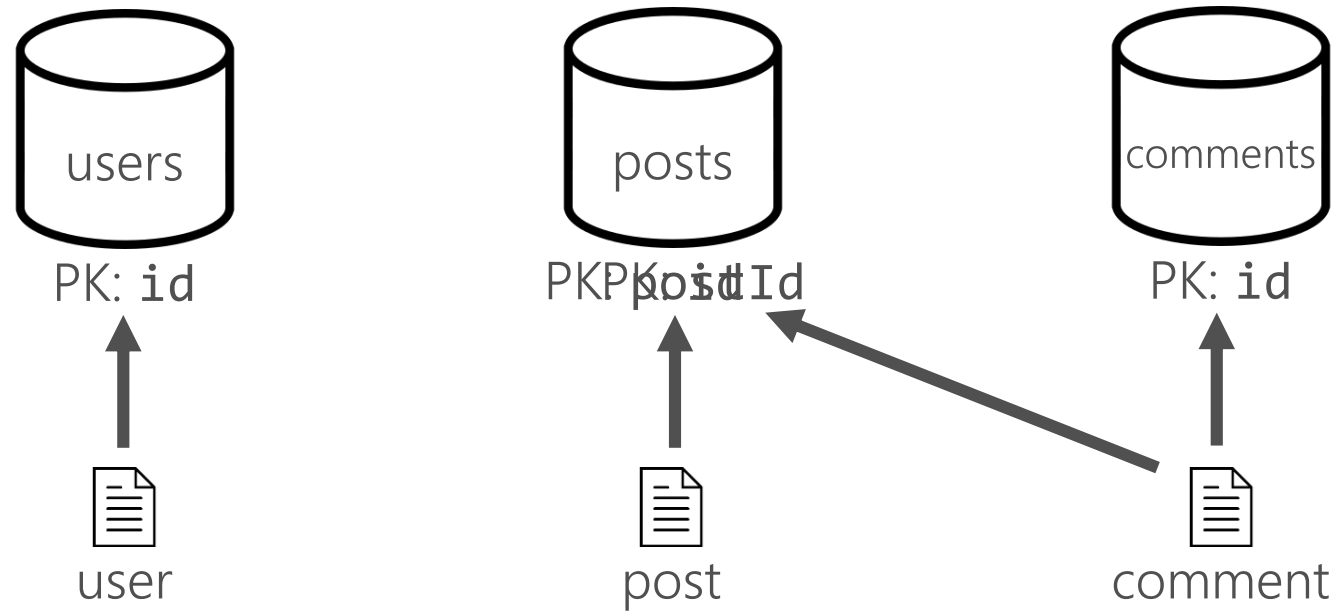
# HOT/COLD PARTITIONS

## PARTITION USAGE CAN VARY OVER TIME

Partitions that are approaching thresholds are referred to as **hot**. Partitions that are underutilized are referred to as **cold**.



# IMPROVING OUR DESIGN BY NOSQL THINKING



- We started with a design where the collection was partitioned by **postId** instead

# MIXING ENTITY TYPES

- Leverage the schema-agnostic nature of Cosmos DB
- Suitable when different entity types
  - share similar access patterns
  - are queried by the same partition key
  - are queried together (for JOIN-like capabilities)
- Typical implementation is to add a "**type**" property to discriminate different types

# REFERENCE VS. EMBED

## Embedding

```
{
  "id": "<post-id>",
  "title": "<post-title>",
  "content": "<post-content>",
  "comments": [
    {
      "id": "<comment-id-1>",
      "content": "..."
    },
    {
      "id": "<comment-id-2>",
      "content": "..."
    }
  ]
}
```

## Referencing

```
{
  "id": "<post-id>",
  "title": "<post-title>",
  "content": "<post-content>"
}
```

```
{
  "id": "<comment-id-1>",
  "postId": "<post-id>",
  "content": "..."
}
```

```
{
  "id": "<comment-id-2>",
  "postId": "<post-id>",
  "content": "..."
}
```

# REFERENCE VS. EMBED

Embed when:

- 1:1
- 1:few
- Items queried together
- Items updated together

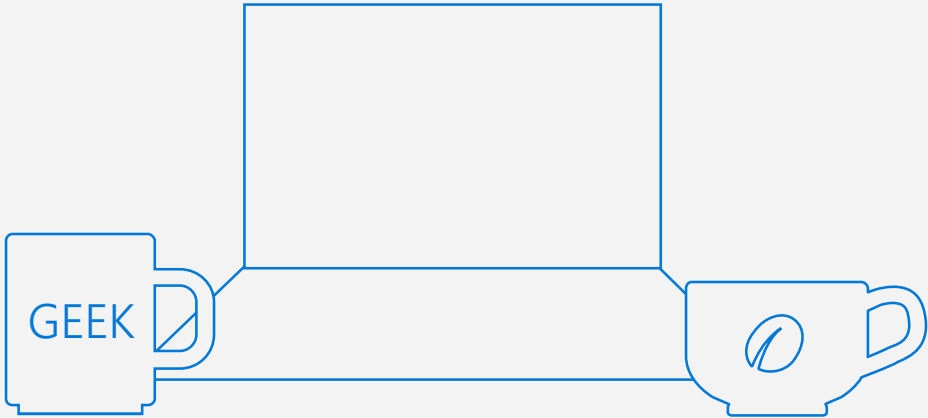
Reference when:

- 1:many (unbounded)
- Many:many
- Items updated independently

# HANDLE ANY DATA WITH NO SCHEMA OR INDEXING REQUIRED

Azure Cosmos DB’s schema-less service automatically indexes all your data, regardless of the data model, to delivery blazing fast queries.

- Automatic index management
- Synchronous auto-indexing
- No schemas or secondary indices needed
- Works across every data model

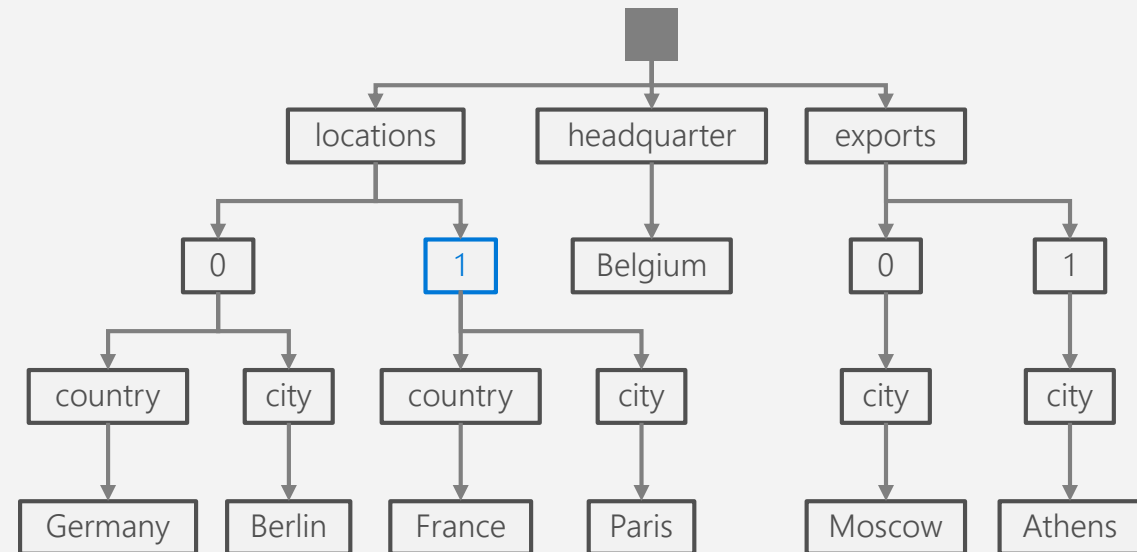


| Item            | Color    | Microwave safe | Liquid capacity | CPU                                 | Memory | Storage  |
|-----------------|----------|----------------|-----------------|-------------------------------------|--------|----------|
| Geek mug        | Graphite | Yes            | 16ox            | ???                                 | ???    | ???      |
| Coffee Bean mug | Tan      | No             | 12oz            | ???                                 | ???    | ???      |
| Surface book    | Gray     | ???            | ???             | 3.4 GHz Intel Skylake Core i7-6600U | 16GB   | 1 TB SSD |



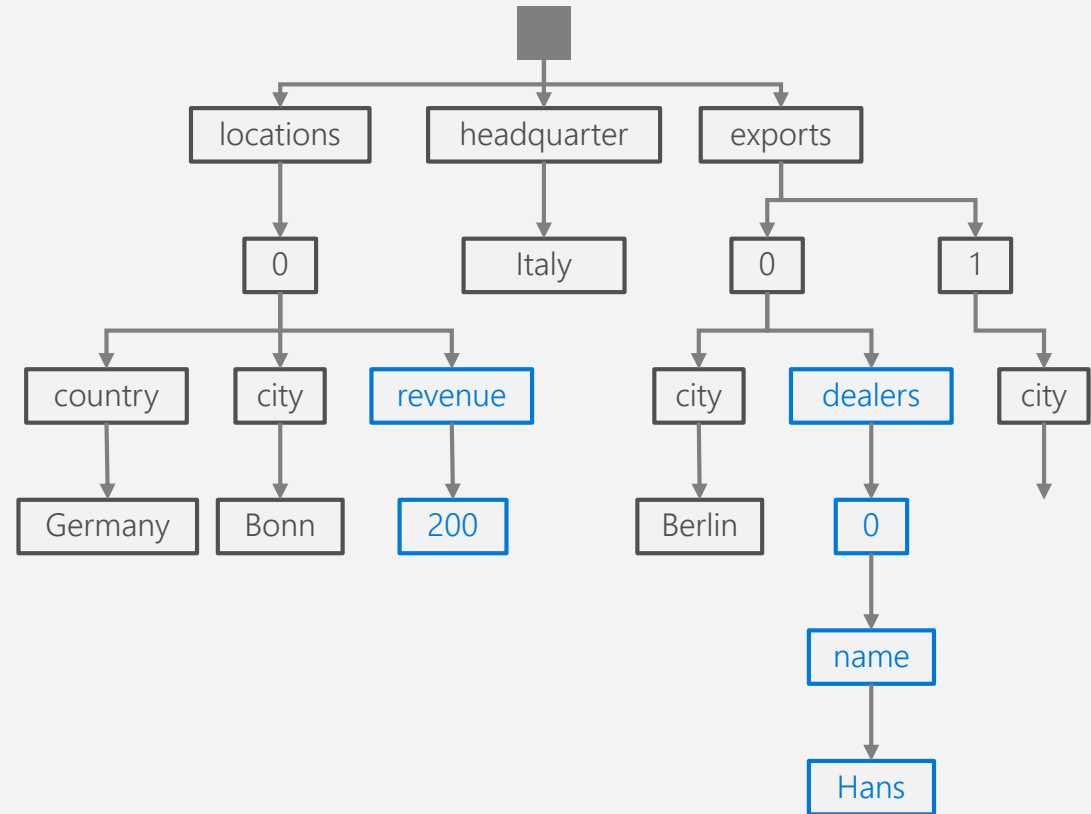
# INDEXING JSON DOCUMENTS

```
{  
  "locations": [  
    {  
      "country": "Germany",  
      "city": "Berlin"  
    },  
    {  
      "country": "France",  
      "city": "Paris"  
    }  
  ],  
  "headquarter": "Belgium",  
  "exports": [  
    { "city": "Moscow" },  
    { "city": "Athens" }  
  ]  
}
```

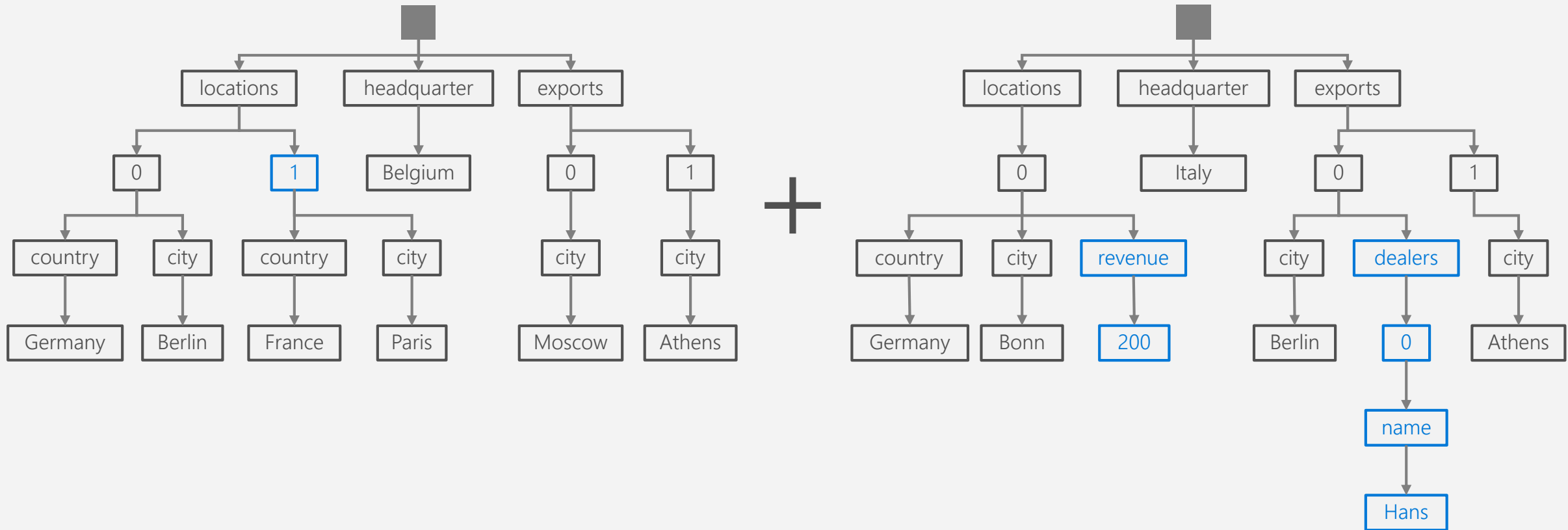


# INDEXING JSON DOCUMENTS

```
{
  "locations": [
    {
      "country": "Germany",
      "city": "Bonn",
      "revenue": 200
    }
  ],
  "headquarter": "Italy",
  "exports": [
    {
      "city": "Berlin",
      "dealers": [
        { "name": "Hans" }
      ]
    },
    { "city": "Athens" }
  ]
}
```



# INDEXING JSON DOCUMENTS



# INDEX POLICIES

## CUSTOM INDEXING POLICIES

Though all Azure Cosmos DB data is indexed by default, you can specify a custom indexing policy for your collections. Custom indexing policies allow you to design and customize the shape of your index while maintaining schema flexibility.

- Define trade-offs between storage, write and query performance, and query consistency
- Include or exclude documents and paths to and from the index
- Configure various index types

```
{
  "automatic": true,
  "indexingMode": "Consistent",
  "includedPaths": [{
    "path": "/*",
    "indexes": [{
      "kind": "Hash",
      "dataType": "String",
      "precision": -1
    }, {
      "kind": "Range",
      "dataType": "Number",
      "precision": -1
    }, {
      "kind": "Spatial",
      "dataType": "Point"
    }
  ]
}, {
  "excludedPaths": [{
    "path": "/nonIndexedContent/*"
  }]
}
```

# INDEXING POLICY

```
{
  "indexingMode": "none",
  "automatic": false,
  "includedPaths": [],
  "excludedPaths": []
}
```

No indexing

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/age/?",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "Number",
          "precision": -1
        }
      ]
    },
    {
      "path": "/gender/?",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    }
  ],
  "excludedPaths": [
    {
      "path": "/*"
    }
  ]
}
```

Index some properties

# RANGE INDEXES

These are created by default for each property and are needed for:

**Equality queries:**

```
SELECT * FROM container c WHERE c.property = 'value'
```

**Range queries:**

```
SELECT * FROM container c WHERE c.property > 'value' (works for >, <, >=, <=, !=)
```

**ORDER BY queries:**

```
SELECT * FROM container c ORDER BY c.property
```

**JOIN queries:**

```
SELECT child FROM container c JOIN child IN c.properties WHERE child =  
'value'
```

# SPATIAL INDEXES

These must be added and are needed for geospatial queries:

## **Geospatial distance queries:**

```
SELECT * FROM container c WHERE ST_DISTANCE(c.property, { "type": "Point",  
"coordinates": [0.0, 10.0] }) < 40
```

## **Geospatial within queries:**

```
SELECT * FROM container c WHERE ST_WITHIN(c.property, {"type": "Point",  
"coordinates": [0.0, 10.0] } })
```

# COMPOSITE INDEXES

**These must be added and are needed for queries that ORDER BY two or more properties.**

**ORDER BY queries on multiple properties:**

```
SELECT * FROM container c ORDER BY c.firstName, c.lastName
```

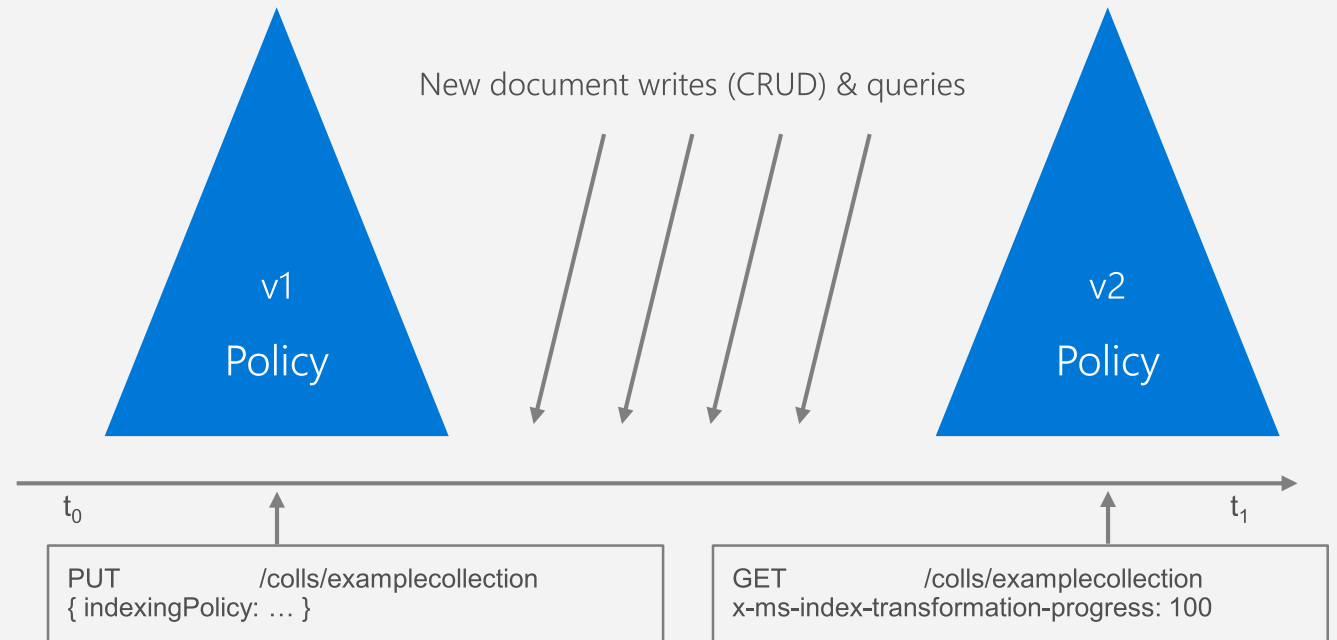


# ONLINE INDEX TRANSFORMATIONS

## On-the-fly Index Changes

In Azure Cosmos DB, you can make changes to the indexing policy of a collection on the fly. Changes can affect the shape of the index, including paths, precision values, and its consistency model.

A change in indexing policy effectively requires a transformation of the old index into a new index.



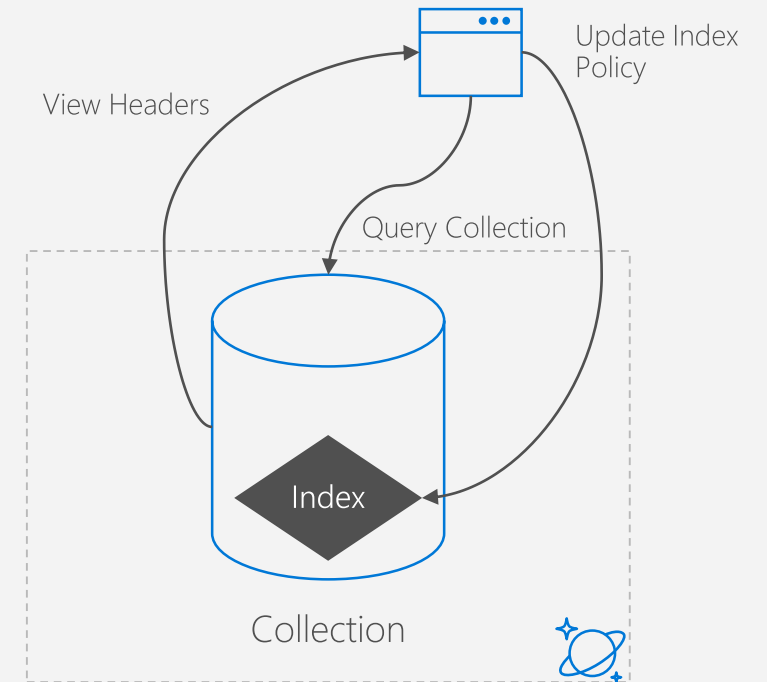
# INDEX TUNING

## Metrics Analysis

The SQL APIs provide information about performance metrics, such as the index storage used and the throughput cost (request units) for every operation. You can use this information to compare various indexing policies, and for performance tuning.

When running a HEAD or GET request against a collection resource, the `x-ms-request-quota` and the `x-ms-request-usage` headers provide the storage quota and usage of the collection.

You can use this information to compare various indexing policies, and for performance tuning.



# BEST PRACTICES

By default is good but not best

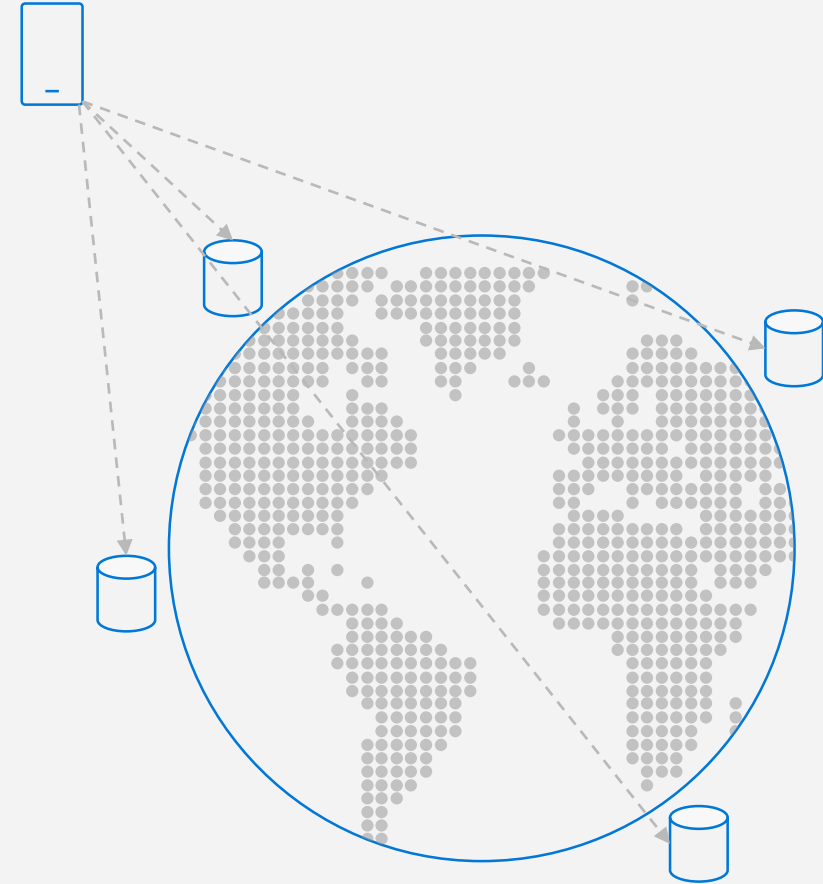
Understand query patterns – which properties are being used?

Understand impact on write cost – index update RU cost scales with # properties

# QUERYING

Tuning query techniques and parameters to make the most efficient use of a globally distributed database service.

*This module will reference querying in the context of the SQL API for Azure Cosmos DB.*



# QUERY TUNING

**MULTIPLE THINGS CAN IMPACT THE PERFORMANCE OF A QUERY RUNNING IN AZURE COSMOS DB. A FEW IMPORTANT QUERY PERFORMANCE FACTORS INCLUDE:**

## **Provisioned throughput**

Measure RU per query, and ensure that you have the required provisioned throughput for your queries

## **Partitioning and partition keys**

Favor queries with the partition key value in the filter clause for low latency

## **SDK and query options**

Follow SDK best practices like direct connectivity, and tune client-side query execution options

# QUERY TUNING

**MANY THINGS CAN IMPACT THE PERFORMANCE OF A QUERY RUNNING IN AZURE COSMOS DB. IMPORTANT PERFORMANCE FACTORS INCLUDE:**

## **Network latency**

Account for network overhead in measurement, and use multi-homing APIs to read from the nearest region

## **Indexing Policy**

Ensure that you have the required indexing paths/policy for the query

## **Query Complexity**

Use simple queries to enable greater scale.

## **Query execution metrics**

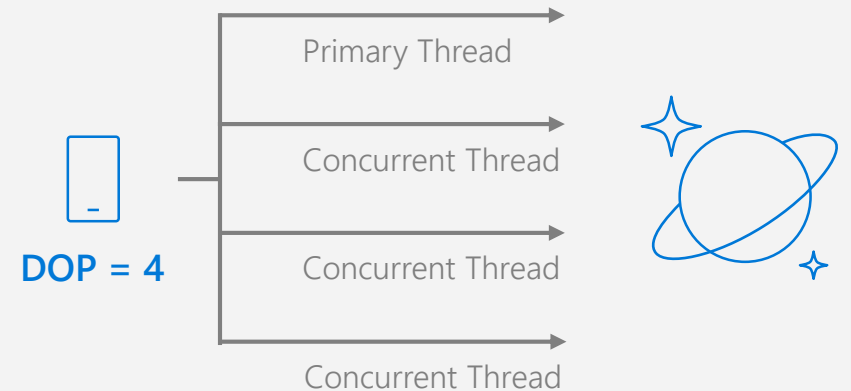
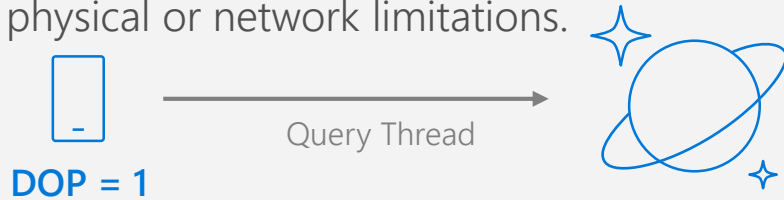
Analyze the query execution metrics to identify potential rewrites of query and data shapes

# CLIENT QUERY PARALLELISM

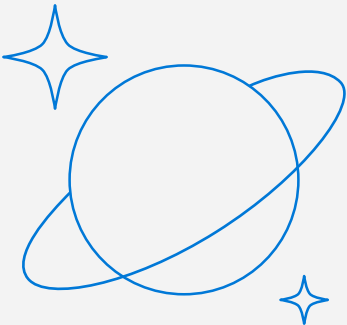
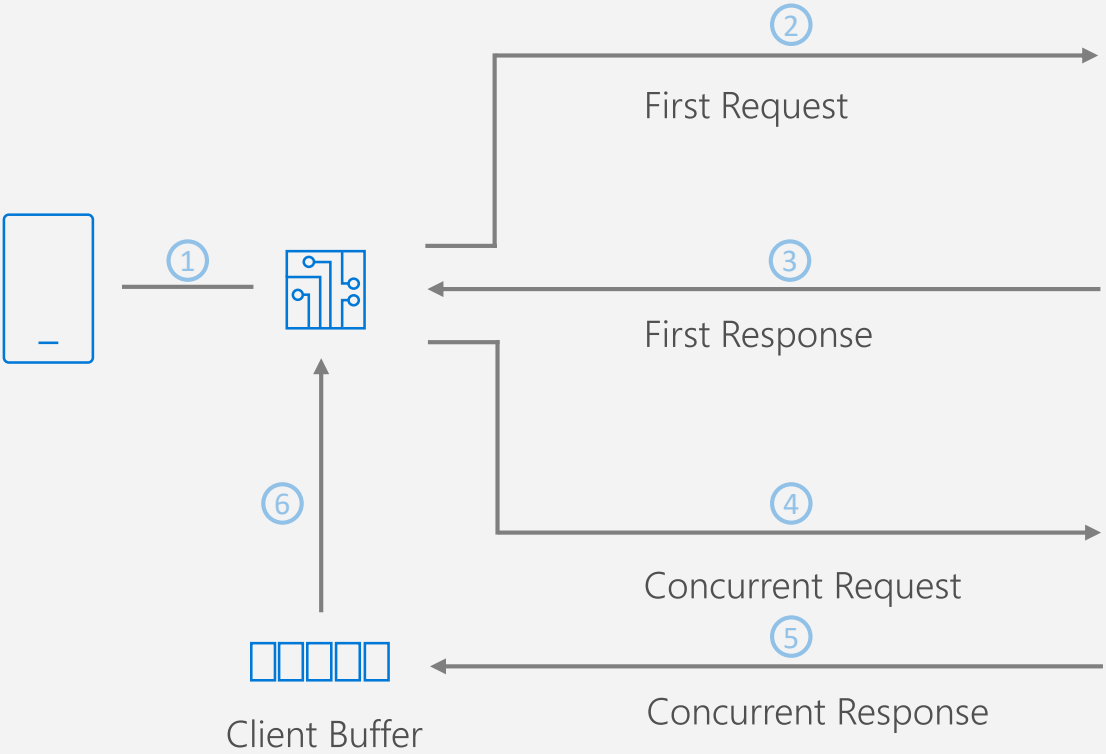
## CROSS-PARTITION QUERIES CAN BE PARALLELIZED TO USE AS MANY THREADS AS POSSIBLE

Modern processors ship with both physical and virtual (hyper-threading) cores. For any given cross-partition query, the SDK can use concurrent threads to issue the query across the underlying partitions.

By default, the SDK uses a **slow start algorithm** for cross-partition queries, increasing the amount of threads over time. This increase is exponential up to any physical or network limitations.



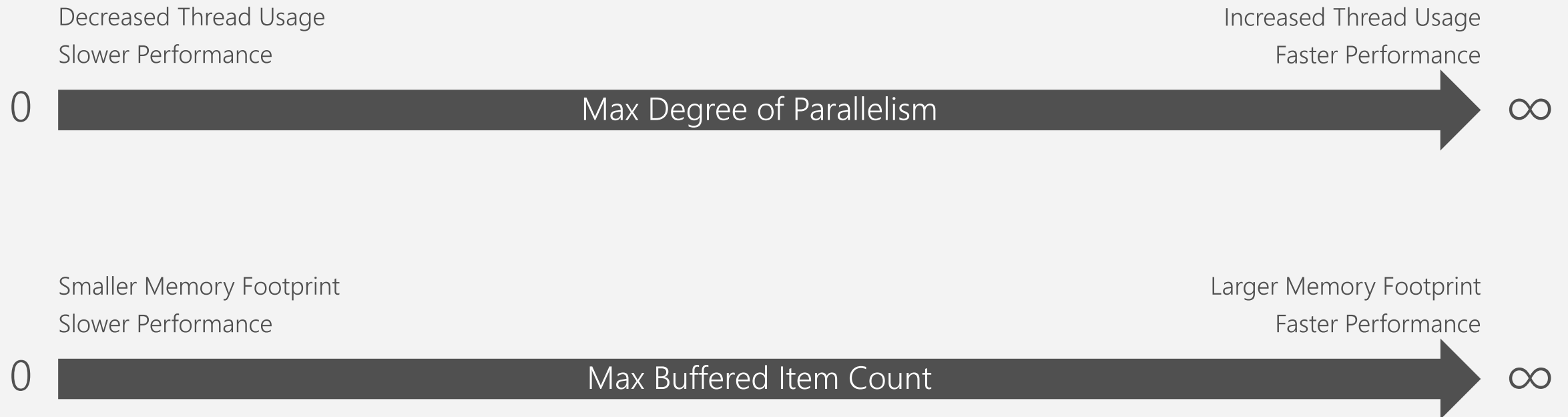
# CLIENT RESPONSE BUFFER





# SDK QUERY OPTIONS

ACHIEVING OPTIMAL PERFORMANCE IS OFTEN A BALANCING ACT  
BETWEEN THESE TWO PROPERTIES



# SDK QUERY OPTIONS

| Setting                | Value | Effect  |
|------------------------|-------|---|
| MaxDegreeofParallelism | -1    | The system will automatically decide the number of items to buffer              |
|                        | 0     | Do not add any additional concurrent threads                                    |
|                        | >= 1  | Add the specified number of additional concurrent threads                       |
| MaxBufferedItemCount   | -1    | The system will automatically decide the number of concurrent operations to run |
|                        | 0     | Do not maintain a client-side buffer  |
|                        | >= 1  | Specify the maximum size (items) of the client-side buffer                      |

# MEASURING RU CHARGE

## ANALYZE QUERY COMPLEXITY

The complexity of a query impacts how many Request Units are consumed for an operation. The number of predicates, nature of the predicates, number of system functions, and the number of index matches / query results all influence the cost of query operations.

## MEASURE QUERY COST

To measure the cost of any operation (create, update, or delete):

- Inspect the x-ms-request-charge header
- Inspect the RequestCharge property in ResourceResponse or FeedResponse in the SDK

## NUMBER OF INDEXED TERMS IMPACTS WRITE RU CHARGES

Every write operation will require the indexer to run. The more indexed terms you have, the more indexing will be directly having an effect on the RU charge.

You can optimize for this by fine-tuning your index policy to include only fields and/or paths certain to be used in queries.

# MEASURING RU CHARGE

## STABILIZED LOGICAL CHARGES

Azure Cosmos DB uses information about past runs to produce a stable logical charge for the majority of CRUD or query operations.

Since this stable charge exists, we can rely on our operations having a **high degree of predictability** with very little variation. We can use the predictable RU charges for future capacity planning.

## BULK OF QUERY RU CHARGES IS IO

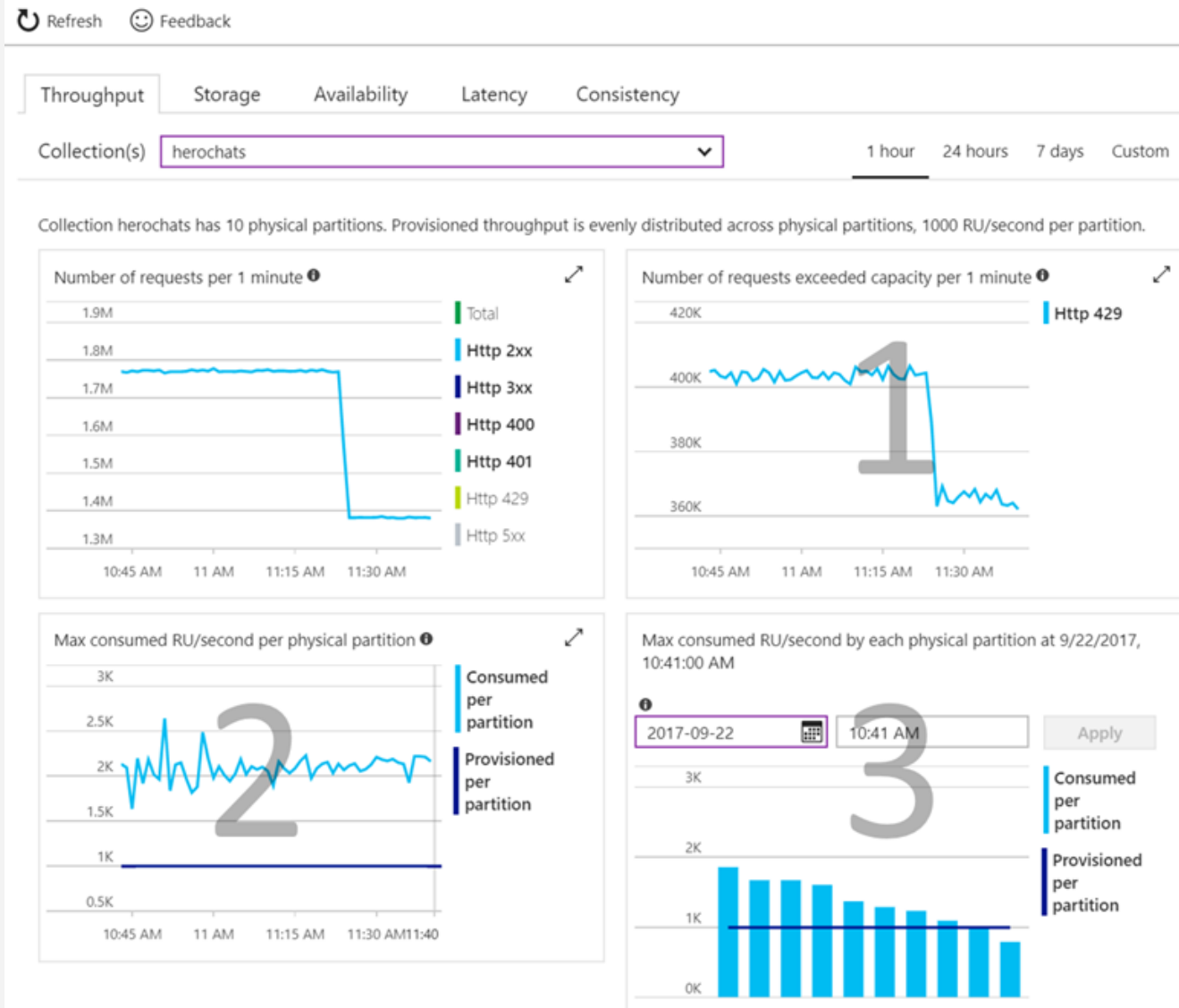
Query RU is directly proportional to the quantity of query results.

DEMO

# Tuning a Query

# VALIDATING THROUGHPUT LEVEL CHOICE

1. Check if your operations are getting rate limited.
  - Requests exceeding capacity chart
2. Check if consumed throughput exceeds the provisioned throughput on any of the physical partitions
  - Max RU/second consumed per partition chart
3. Select the time where the maximum consumed throughput per partition exceeded provisioned on the chart
  - Max consumed throughput by each partition chart

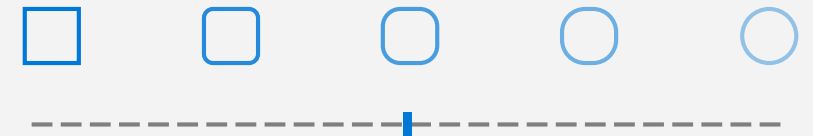
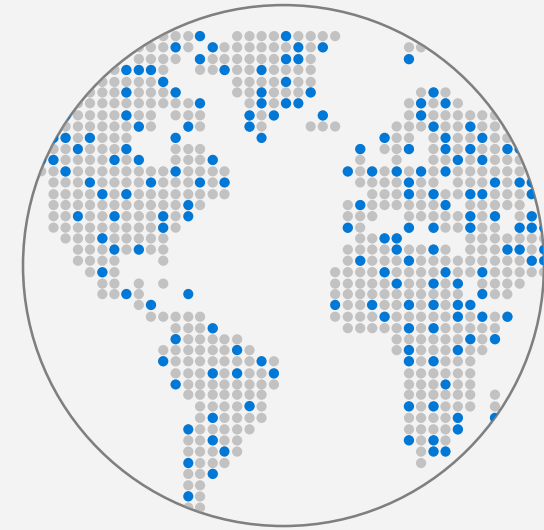


# Other TIPS and TRICKS

# MODELING & PLANNING

Modeling data & configuring containers to take advantage of Azure Cosmos DB strengths.

*This module will reference modeling in the context of all Azure Cosmos DB modules and APIs.*





# CONTAINERS

## CONTAINERS

- Containers do NOT enforce schema
- There are benefits to co-locate multiple types in a container
- Annotate records with a "type" property

## CO-LOCATING TYPES IN THE SAME CONTAINER

- Ability to query across multiple entity types with a single network request.
- Ability to perform transactions across multiple types
- Cost: reduce physical partition footprint

# CO-LOCATING TYPES

Ability to query across multiple entity types with a single network request.

For example, we have two types of documents: cat and person.

```
{  
  "id": "Andrew",  
  "type": "Person",  
  "familyId": "Liu",  
  "worksOn": "Azure Cosmos DB"  
}
```

```
{  
  "id": "Ralph",  
  "type": "Cat",  
  "familyId": "Liu",  
  "fur": {  
    "length": "short",  
    "color": "brown"  
  }  
}
```

We can query both types of documents without needing a JOIN simply by running a query without a filter on type:

```
SELECT * FROM c WHERE c.familyId = "Liu"
```

If we wanted to filter on type = "Person", we can simply add a filter on type to our query:

```
SELECT * FROM c WHERE c.familyId = "Liu" AND c.type = "Person"
```

# UPDATING NORMALIZED DATA

## UPDATES ARE ATOMIC

Update operations update the entire document, not specific fields or “parts” of the document.

## DE-NORMALIZED DOCUMENTS CAN BE EXPENSIVE TO UPDATE

De-normalization has benefits for read operations, but you must weigh this against the costs in write operations.

De-normalization may require fanning out update operations.

Normalization may require chaining a series of requests to resolve relationships.

```
{
  "id": "08259",
  "ticketPrice": 255.00,
  "flightCode": "3754",
  "origin": {
    "airport": "SEA",
    "gate": "A13",
    "departure": "2014-09-15T23:14:25.7251173Z"
  },
  "destination": {
    "airport": "JFK",
    "gate": "D4",
    "arrival": "2014-09-16T02:10:10.2379581Z"
  },
  "pilot": [{
    "id": "EBAMAO",
    "name": "Hailey Nelson"
  }]
}
```

# UPDATING NORMALIZED DATA

## Normalized: Optimized for writes over reads

```
{
  "id": "08259",
  "pilot": [{ "id": "EBAMAO", "name": "Hailey Nelson" }]
},
{
  "id": "08259",
  "ticketPrice": 255.00,
  "flightCode": "3754"
},
{
  "id": "08259",
  "origin": {
    "airport": "SEA", "gate": "A13",
    "departure": "2014-09-15T23:14:25.7251173Z"
  },
  "destination": {
    "airport": "JFK", "gate": "D4",
    "arrival": "2014-09-16T02:10:10.2379581Z"
  }
}
```

## De-normalized: Optimized for reads over writes

```
{
  "id": "08259",
  "ticketPrice": 255.00,
  "flightCode": "3754",
  "origin": {
    "airport": "SEA",
    "gate": "A13",
    "departure": "2014-09-15T23:14:25.7251173Z"
  },
  "destination": {
    "airport": "JFK",
    "gate": "D4",
    "arrival": "2014-09-16T02:10:10.2379581Z"
  },
  "pilot": [{
    "id": "EBAMAO",
    "name": "Hailey Nelson"
  }]
}
```

# UPDATING NORMALIZED DATA

## THE SOLUTION IS TYPICALLY A COMPROMISE BASED ON YOUR WORKLOAD

Examine your workload. Answer the following questions:

- Which fields are commonly updated together?
- What are the most common fields included in all queries?

**Example:** The ticketPrice, origin and destination fields are often updated together. The pilot field is only rarely updated. The flightCode field is included in almost all queries across the board.

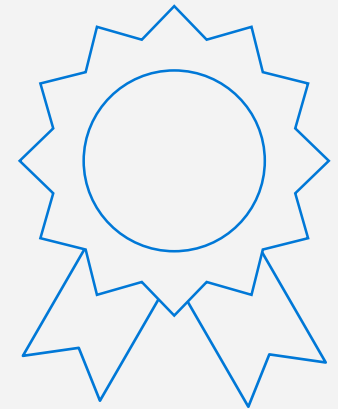
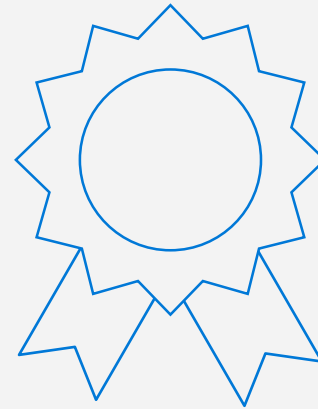
```
{
  "id": "08259",
  "flightCode": "3754",
  "pilot": [{ "id": "EBAMAO", "name": "Hailey Nelson" }]
},
{
  "id": "08259",
  "flightCode": "3754",
  "ticketPrice": 255.00,
  "origin": {
    "airport": "SEA", "gate": "A13",
    "departure": "2014-09-15T23:14:25.7251173Z"
  },
  "destination": {
    "airport": "JFK", "gate": "D4",
    "arrival": "2014-09-16T02:10:10.2379581Z"
  }
}
```

# SHORT-LIFETIME DATA

Some data produced by applications are only useful for a finite period of time:

- Machine-generated event data
- Application log data
- User session information

It is important that the database system systematically purges this data at pre-configured intervals.



# TIME-TO-LIVE (TTL)

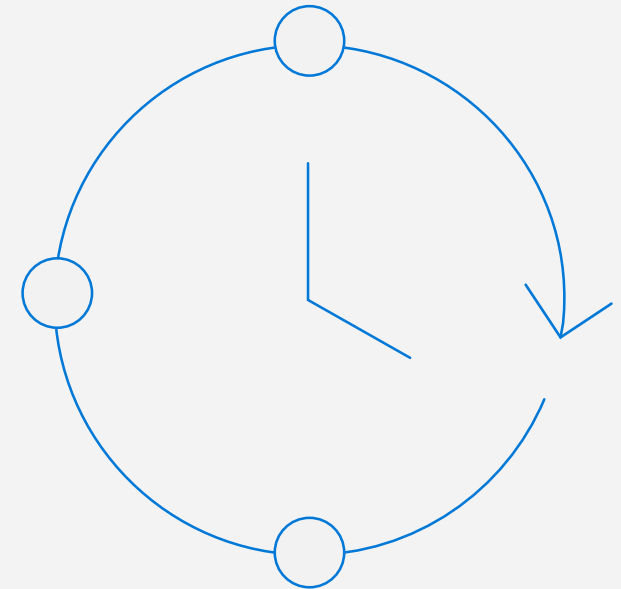
## AUTOMATICALLY PURGE DATA

Azure Cosmos DB allows you to set the length of time in which documents live in the database before being automatically purged. A document's "time-to-live" (TTL) is measured in seconds from the last modification and can be set at the collection level with override on a per-document basis.

The TTL value is specified in the `_ts` field which exists on every document.

- The `_ts` field is a unix-style epoch timestamp representing the date and time. The `_ts` field is updated every time a document is modified.

Once TTL is set, Azure Cosmos DB will automatically remove documents that exist after that period of time.



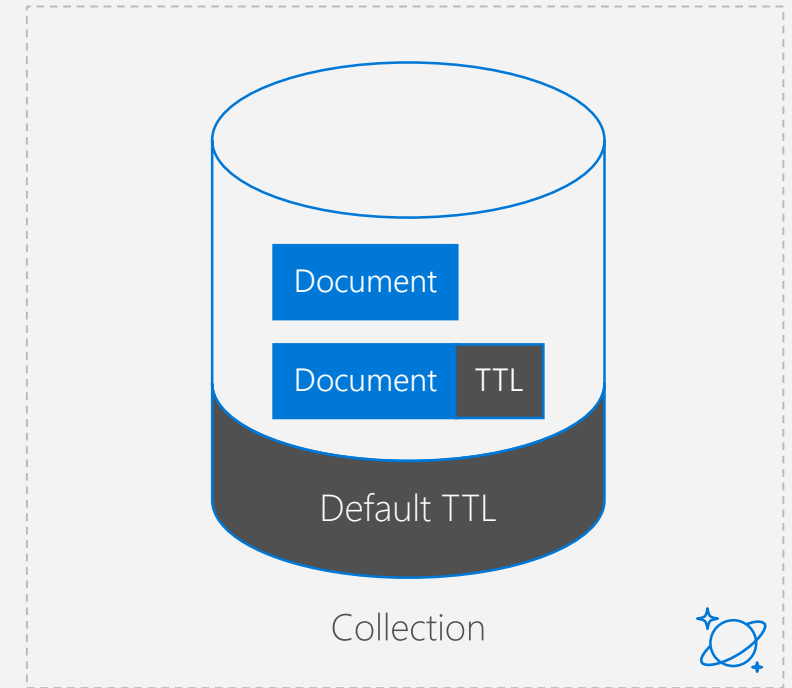
# EXPIRING RECORDS USING TIME-TO-LIVE

## TTL BEHAVIOR

The TTL feature is controlled by TTL properties at two levels - the collection level and the document level.

- DefaultTTL for the collection
  - If missing (or set to null), documents are not deleted automatically.
  - If present and the value is "-1" = infinite – documents don't expire by default
  - If present and the value is some number ("n") – documents expire "n" seconds after last modification
- TTL for the documents:
  - Property is applicable only if DefaultTTL is present for the parent collection.
  - Overrides the DefaultTTL value for the parent collection.

The values are set in seconds and are treated as a delta from the `_ts` that the document was last modified at.





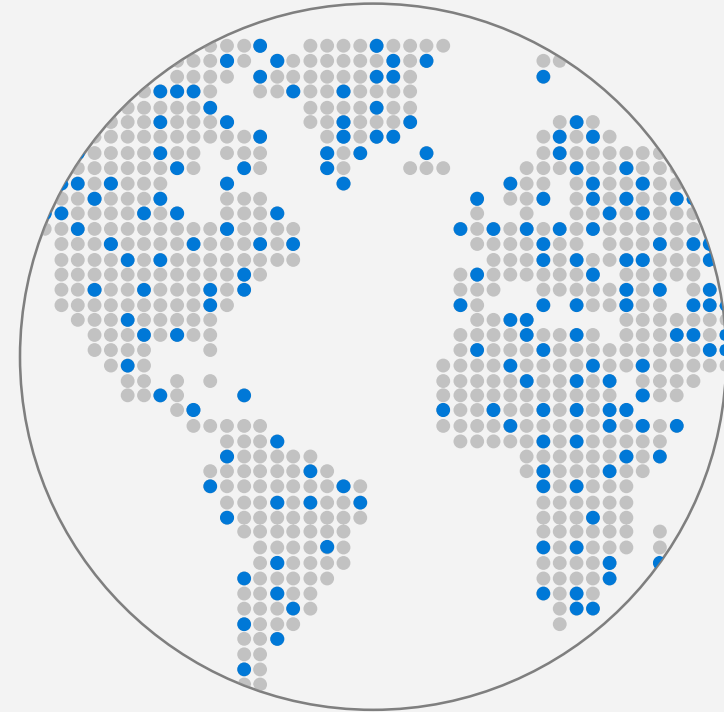
# TURNKEY GLOBAL DISTRIBUTION

## High Availability

- Automatic and Manual Failover
- Multi-homing API removes need for app redeployment

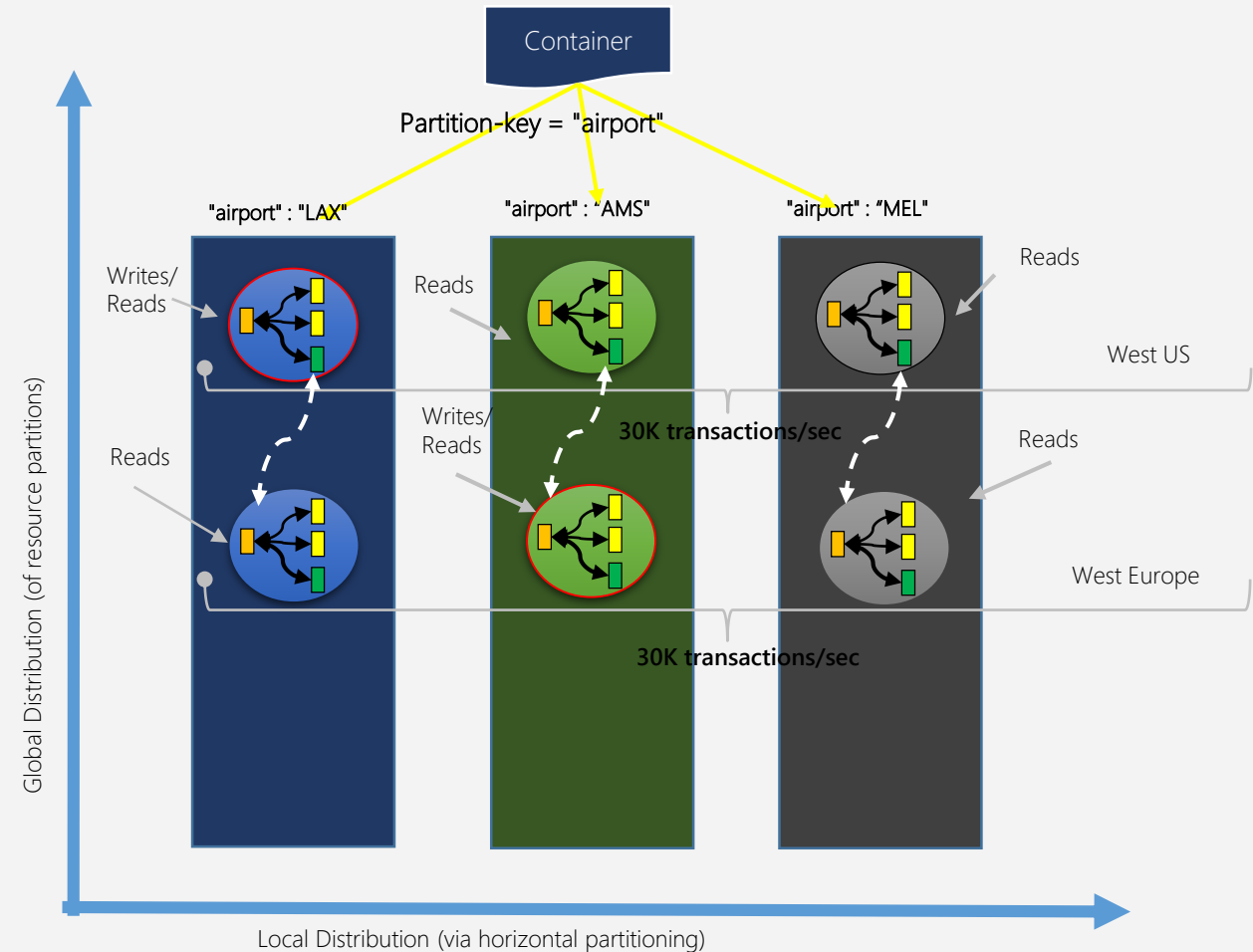
## Low Latency (anywhere in the world)

- Packets cannot move faster than the speed of light
- Sending a packet across the world under ideal network conditions takes 100's of milliseconds
- You can cheat the speed of light – using data locality
  - CDN's solved this for static content
  - Azure Cosmos DB solves this for dynamic content



# TURNKEY GLOBAL DISTRIBUTION

- Automatic and transparent replication worldwide
- Each partition hosts a replica set per region
- Customers can test end to end application availability by programmatically simulating failovers
- All regions are hidden behind a single global URI with multi-homing capabilities
- Customers can dynamically add / remove additional regions at any time



# REPLICATING DATA GLOBALLY



andrl-global - Replicate data globally

Azure Cosmos DB account

Search (Ctrl+/)

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Quick start

Data Explorer

SETTINGS

Replicate data globally

Default consistency

Firewall

Keys



Save



Discard



Manual Failover



Automatic Failover

Click on a location to add or remove regions from your Azure Cosmos DB account.

\* Each region is billable based on the throughput and storage for the account. [Learn more](#)



# MANUAL FAILOVER

## Manual Failover



Select a Read Region to become the new Write Region.

Tip: Identify all dependent services leveraging this account and ensure that triggering a failover will not jeopardize your production application.

WRITE REGION

Central US

READ REGIONS

Southeast Asia

North Europe



I understand and agree to trigger a failover on my current Write Region.

OK

# TO RESUME AND MY PERSONAL ADVICE

Enable monitoring when you create the endpoint

Cassandra API , mongo API are wrapper , if you start use SQL API

Think at model , organization

- RU / partition

- RU / DB

- usually optimum should be a mix

Partition Key is the key

Avoid default index policy

Think Cloud ( all can run but thinks from the start you pay what you consume ( data , CPU , IO )

Monitor ( DBA is not a "LUXE" )

Other information

# RESPONSE STATUS CODES

| Code                       | Meaning   |
|----------------------------|---|
| <b>200 OK</b>              | GET, PUT or POST operation was successful   |
| <b>201 Created</b>         | Resource created successfully using a POST operation  |
| <b>204 No Content</b>      | Resource deleted successfully using a DELETE operation  |
| <b>401 Unauthorized</b>    | Invalid Authorization header  |
| <b>403 Forbidden</b>       | Authorization token expired<br>Resource quota reached when attempting to create a document<br>Stored Procedure, Trigger or UDF is blacklisted from executed |
| <b>409 Request Timeout</b> | Stored Procedure, Trigger or UDF exceeded maximum execution time  |

# RESPONSE STATUS CODES

| Header                          | Value   |
|---------------------------------|---|
| <b>409 Conflict</b>             | The item Id for a PUT or POST operation conflicts with an existing item                       |
| <b>412 Precondition Failure</b> | The specified eTag is different from the version on the server (optimistic concurrency error) |
| <b>413 Entity Too Large</b>     | The item size exceeds maximum allowable document size of 2MB                                  |
| <b>429 Too Many Requests</b>    | Container has exceeded provisioned throughput limit   |
| <b>449 Retry With</b>           | Transient error has occurred, safe to retry   |
| <b>50x</b>                      | Server-side error. If effort persists, contact support  |



# RESPONSE HEADERS

| Header                       | Value  |
|------------------------------|--|
| <b>x-ms-activity-id</b>      | Unique identifier for the operation                        |
| <b>x-ms-serviceversion</b>   | Service Version used for request/response                  |
| <b>x-ms-schemaversion</b>    | Schema Version used for request/response                   |
| <b>x-ms-item-count</b>       | In a query (or read-feed), the number of items returned    |
| <b>x-ms-alt-content-path</b> | REST URI to access resource using user-supplied IDs        |
| <b>etag</b>                  | The same value as the _etag property of the requested item |

# RESPONSE HEADERS

| Header                     | Value   |
|----------------------------|---|
| <b>x-ms-continuation</b>   | Token returned if a query (or read-feed) has more results and is resubmitted by clients as a request header to resume execution |
| <b>x-ms-session-token</b>  | Used to maintain session consistency. Clients much echo this as a request header in subsequent operations to the same container |
| <b>x-ms-request-charge</b> | Number of normalized RU/s for the operation   |
| <b>x-ms-resource-quota</b> | Allotted quota for the specified resource in the account  |
| <b>x-ms-resource-usage</b> | Current usage count of the specified resource in the account  |
| <b>x-ms-retry-after-ms</b> | If rate limited, the number of milliseconds to wait before retrying the operation   |

