Microsoft

Azure Cosmos DB

Technical Deep Dive

Andre Essing
Technology Solutions Professional
Data Platform

Andre Essing

Technology Solutions Professional
Microsoft Deutschland GmbH

Andre advises customers in topics all around the Microsoft Data Platform. Since version 7.0, Andre gathering experience with the SQL Server product family. Today Andre concentrates on working with data in the cloud, like Modern Data Warehouse architectures, Artificial Intelligence and new scalable database systems like Azure Cosmos DB.

andre.essing@microsoft.com andreessing.de /aessing @aessing aessing

MODERN APPS FACE NEW CHALLENGES

Managing and syncing data distributed around the globe

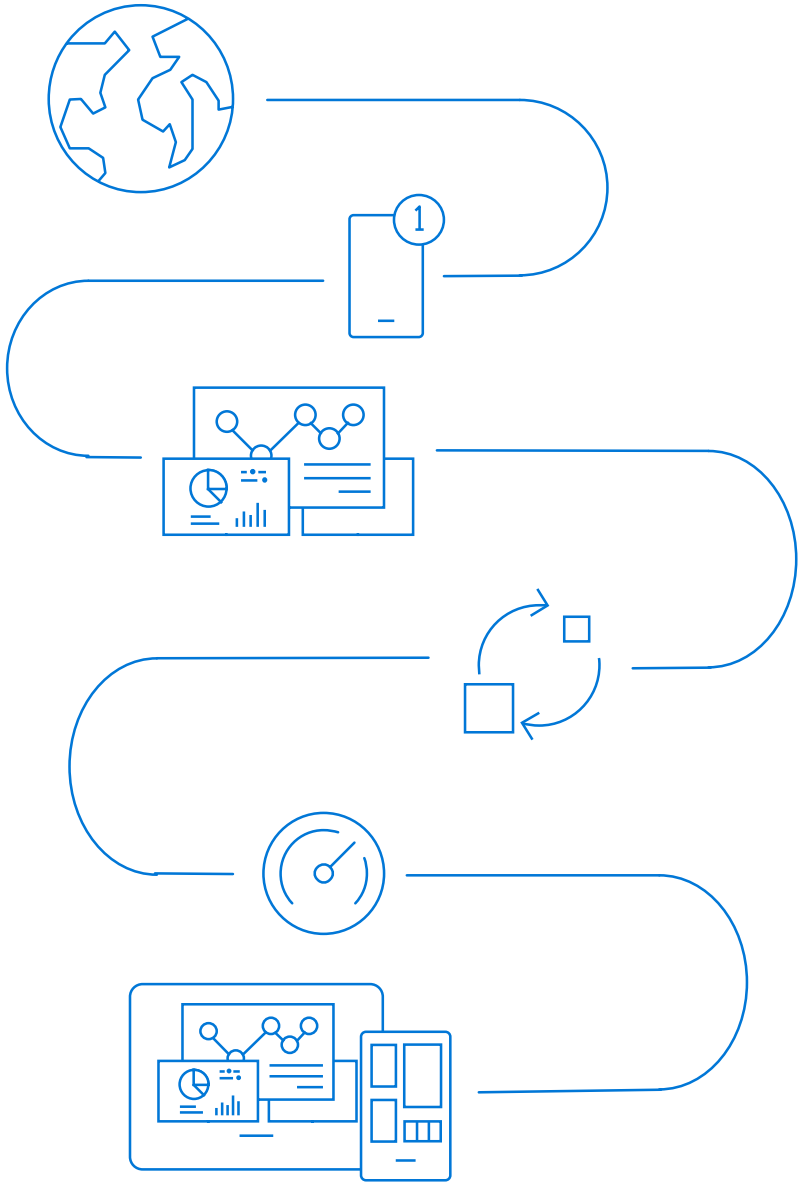
Delivering highly-responsive, real-time personalization

Processing and analyzing large, complex data

Scaling both throughput and storage based on global demand

Offering low-latency to global users

Modernizing existing apps and data

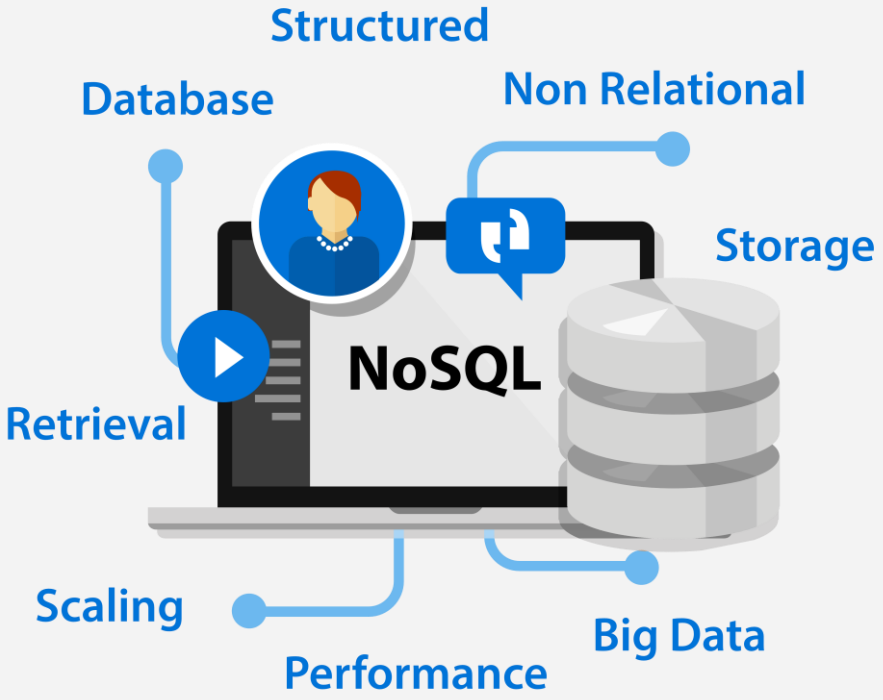


WHAT IS NOSQL

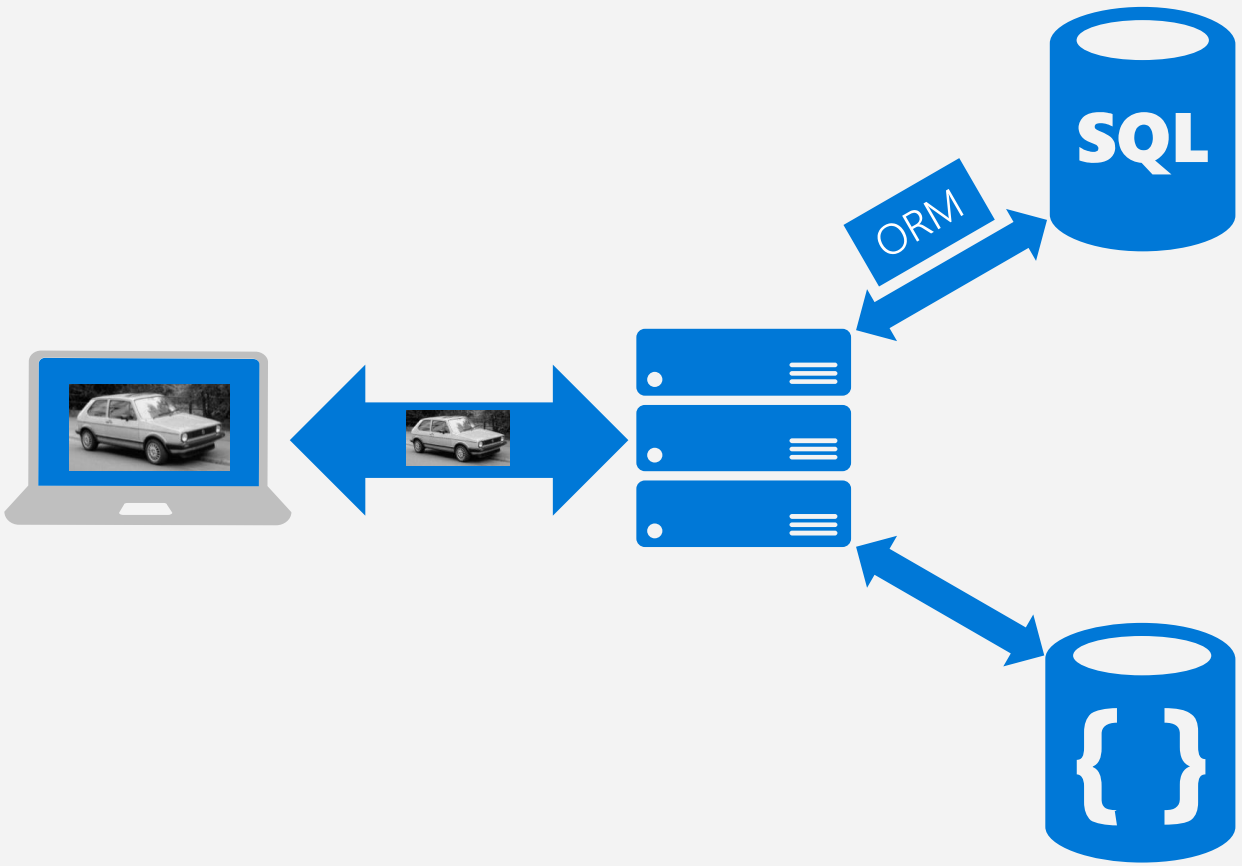
NOSQL, BUILT FOR SIMPLE AND FAST APPLICATION DEVELOPMENT

NoSQL, referring most times to “Non-SQL”, “Not Only SQL” or also “non-relational” is a kind of database where the data is modeled differently to relational systems.

- Different kinds available
  - Document
  - Key/Value
  - Columnar
  - Graph
  - etc.
- Non-Relational
- Schema agnostic
- Built for scale and performance
- Different consistency model



DIFFERENT WAYS OF STORING DATA WITH YOUR MODERN APP

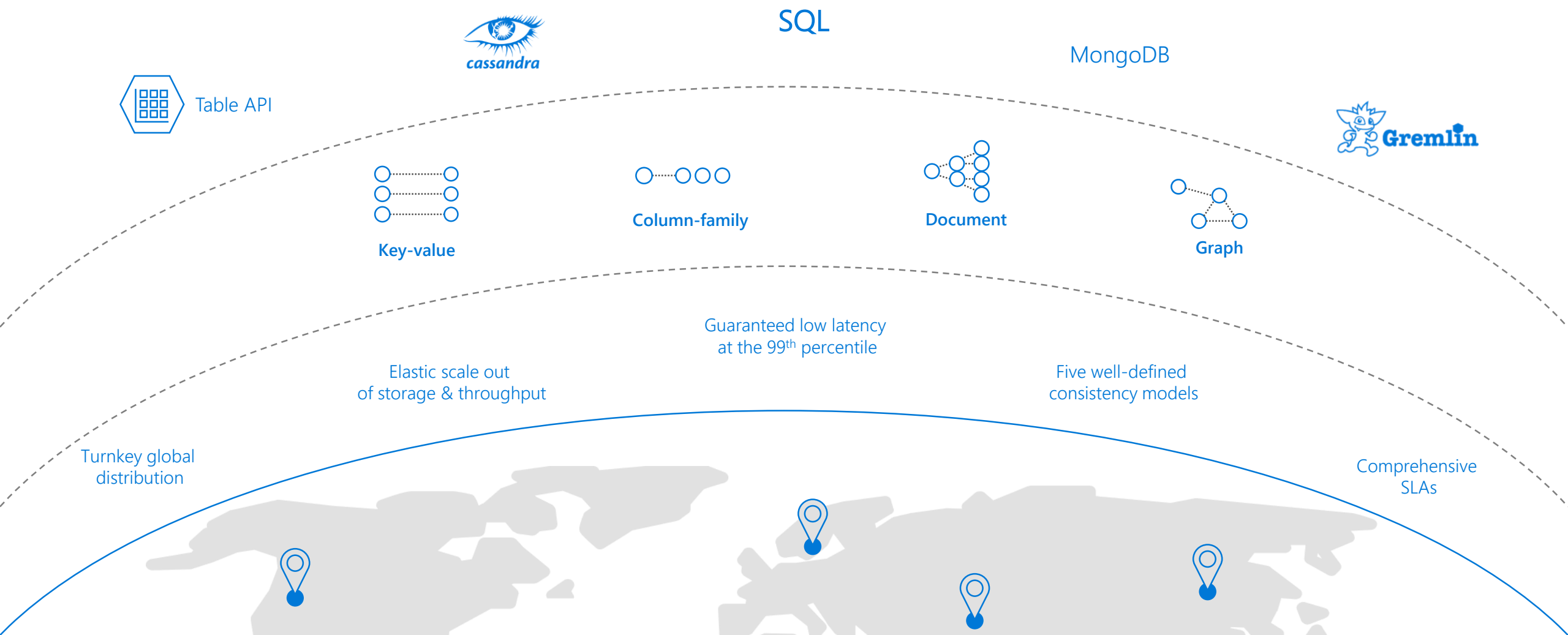


Data normalization

Come as you are

AZURE COSMOS DB

A globally distributed, massively scalable, multi-model database service



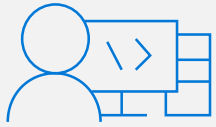
POWERING GLOBAL SOLUTIONS

Azure Cosmos DB was built to support modern app patterns and use cases.

It enables industry-leading organizations to unlock the value of data, and respond to global customers and changing business dynamics in real-time.

Data distributed and available globally

Puts data where your users are

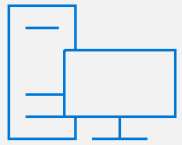


Build real-time customer experiences

Enable latency-sensitive personalization, bidding, and fraud detection.

Ideal for gaming, IoT & eCommerce

Predictable and fast service, even during traffic spikes



Simplified development with serverless architecture

Fully-managed event-driven micro-services with elastic computing power

Run Spark analytics over operational data

Accelerate insights from fast, global data

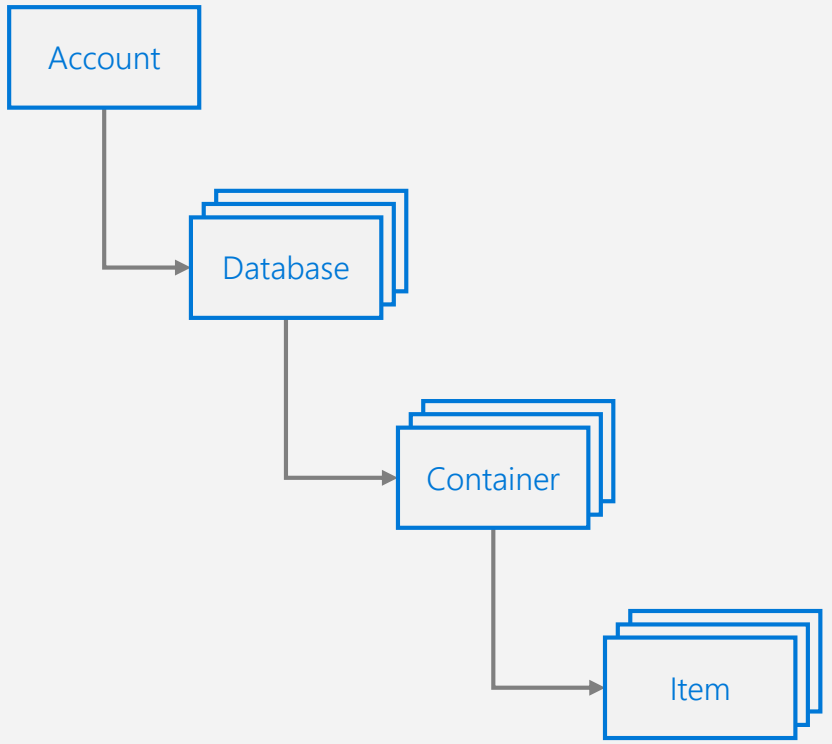
Lift and shift NoSQL data

Lift and shift MongoDB and Cassandra workloads

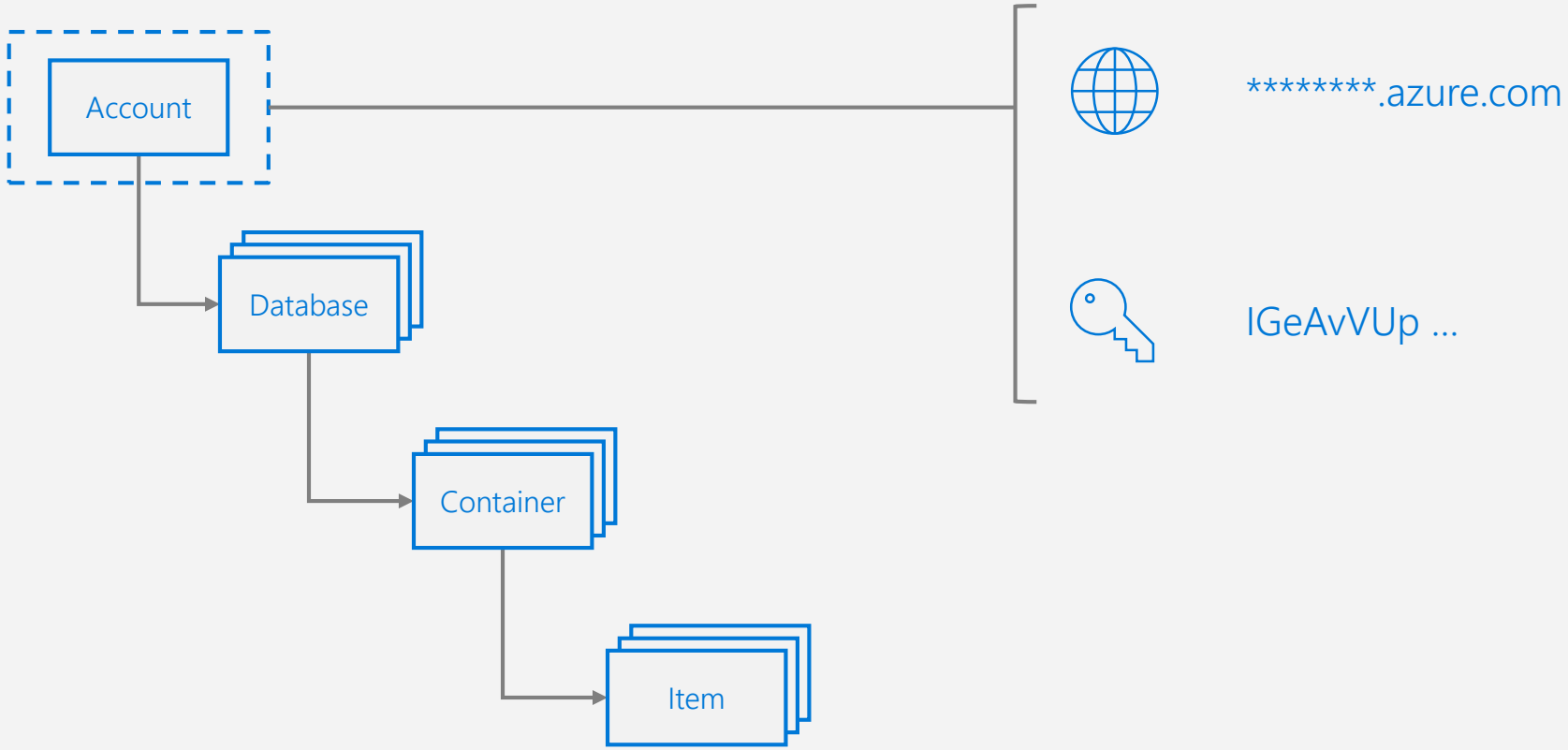
RESOURCE MODEL

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

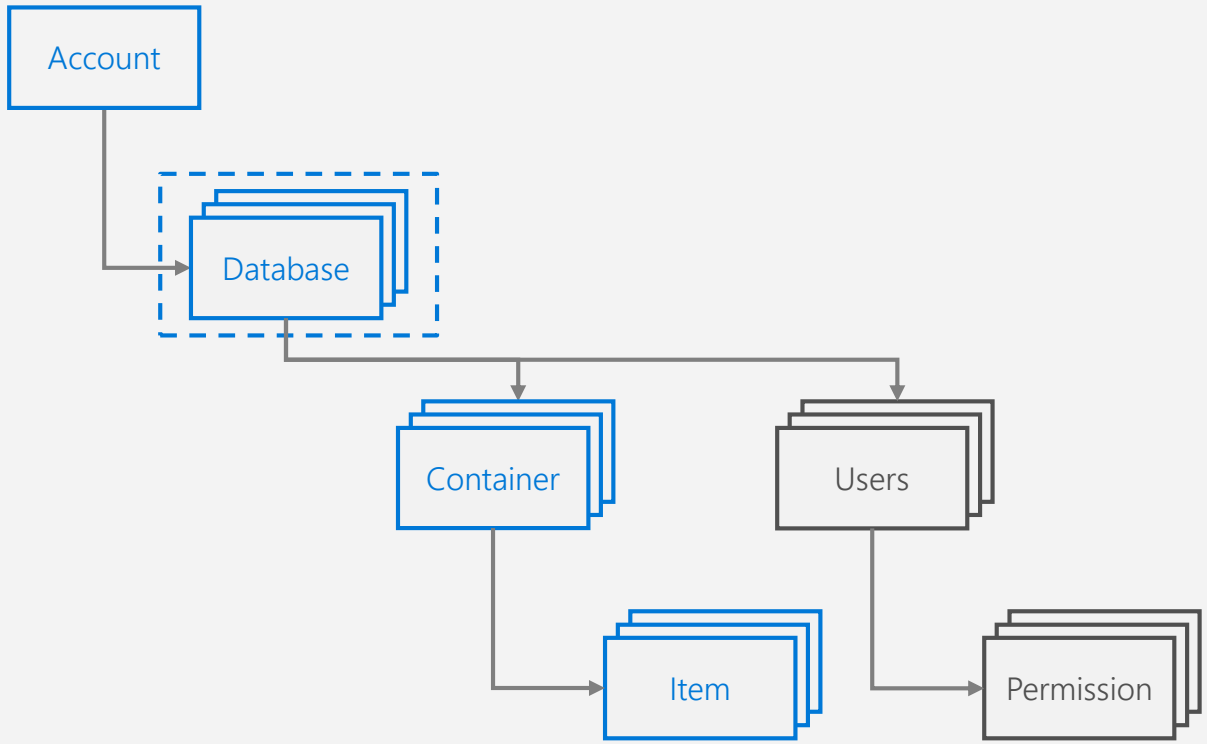
This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.



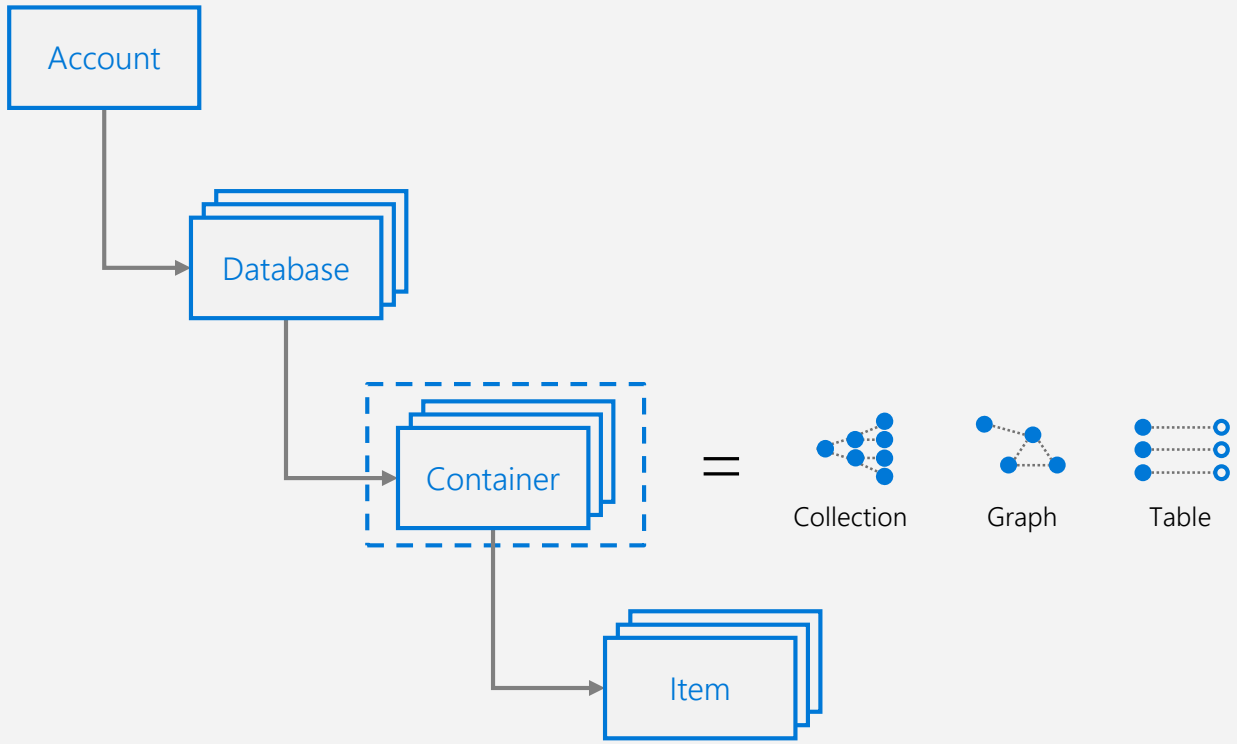
ACCOUNT URI AND CREDENTIALS



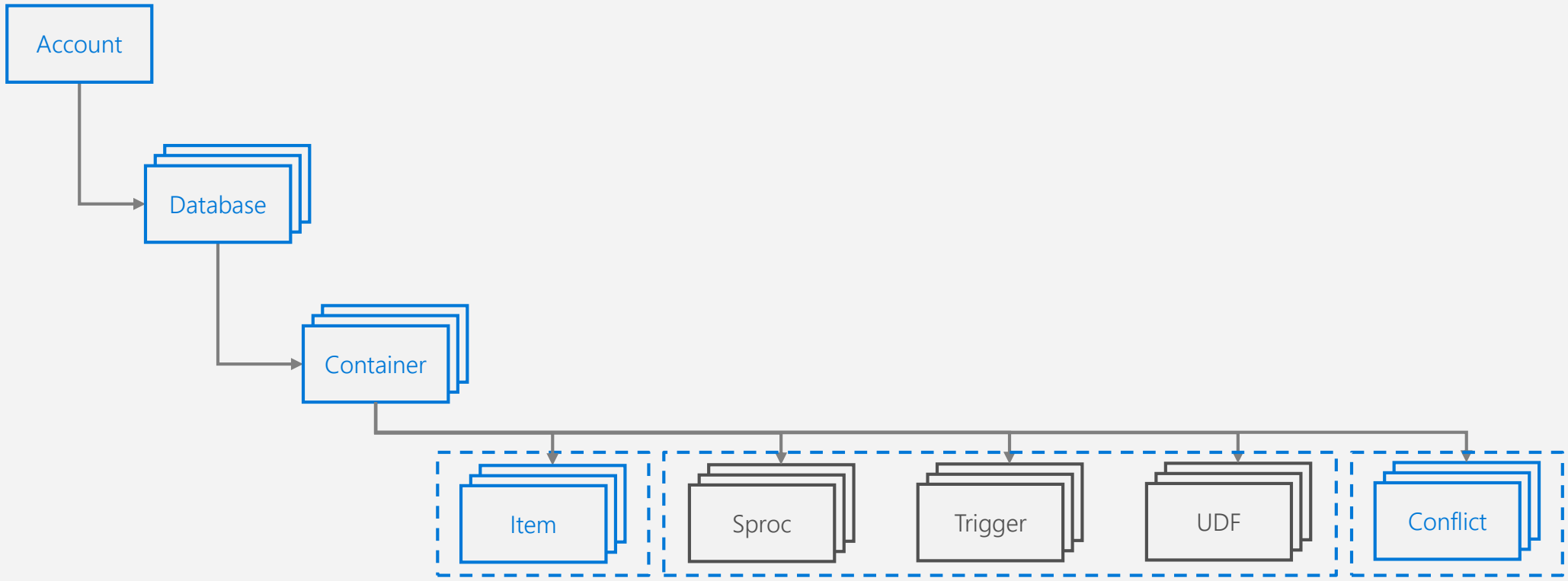
DATABASE REPRESENTATIONS



CONTAINER REPRESENTATIONS



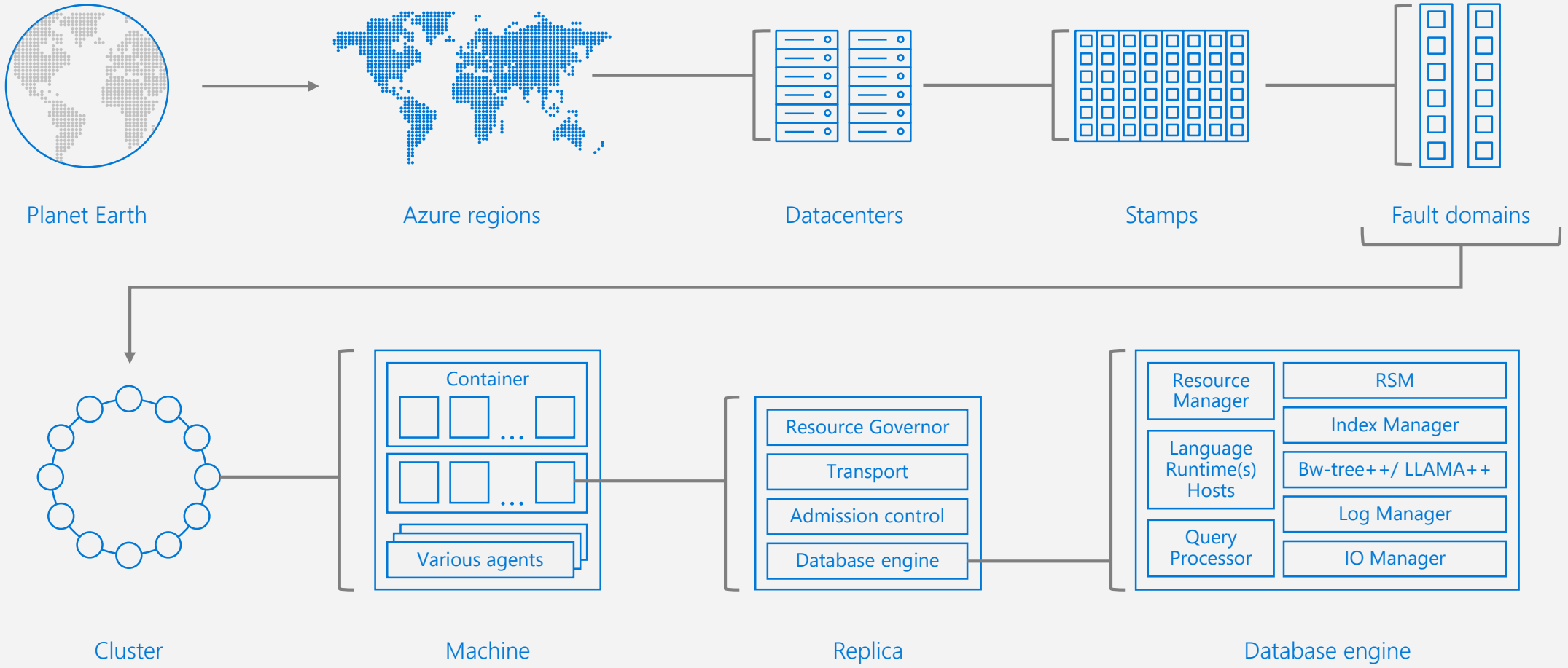
CONTAINER-LEVEL RESOURCES



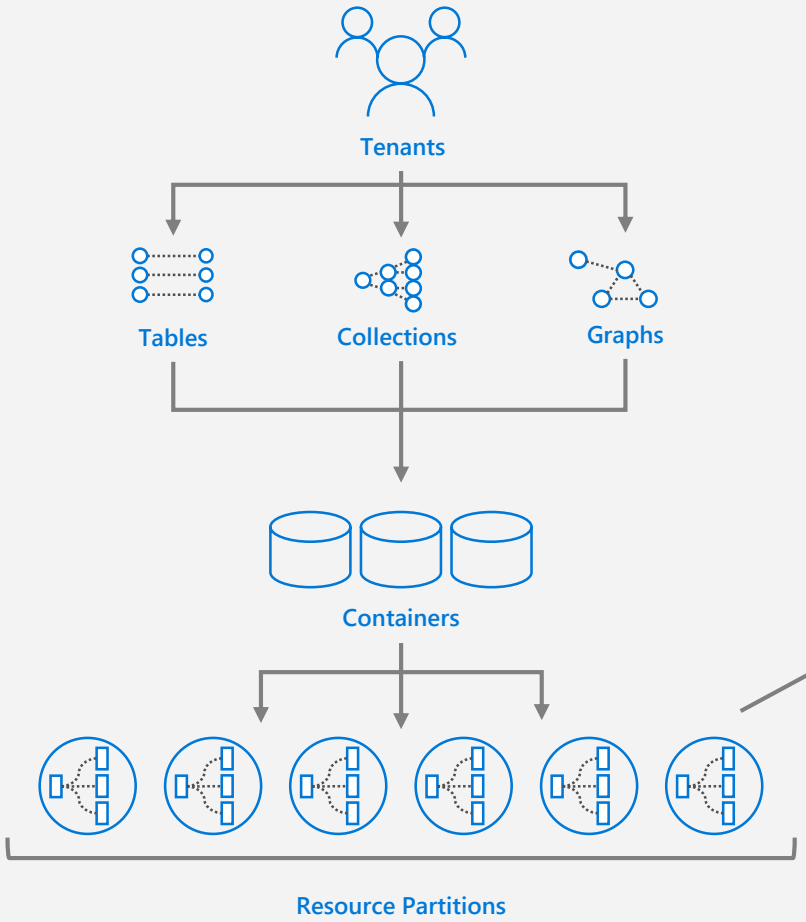
DEMO

Creating a Cosmos DB

SYSTEM TOPOLOGY (BEHIND THE SCENES)

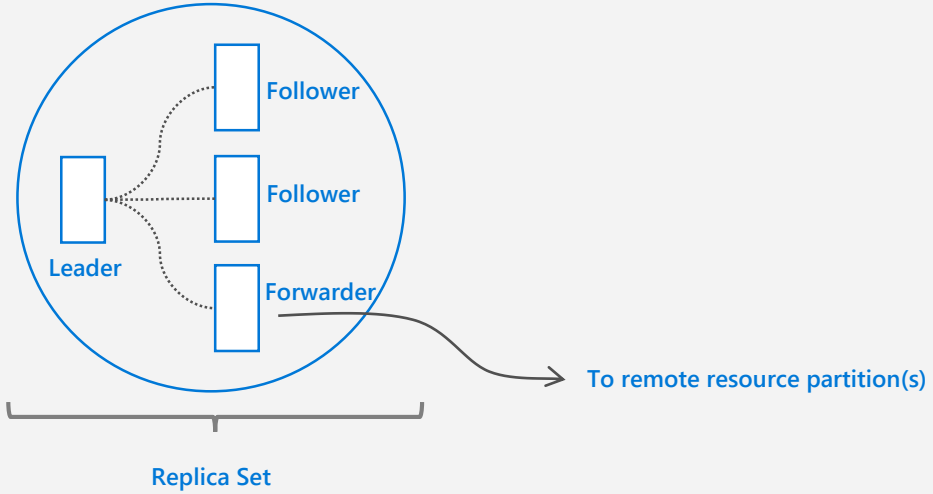


RESOURCE HIERARCHY



CONTAINERS

Logical resources “surfaced” to APIs as tables, collections or graphs, which are made up of one or more physical partitions or servers.



RESOURCE PARTITIONS

- Consistent, highly available, and resource-governed coordination primitives
- Consist of replica sets, with each replica hosting an instance of the database engine

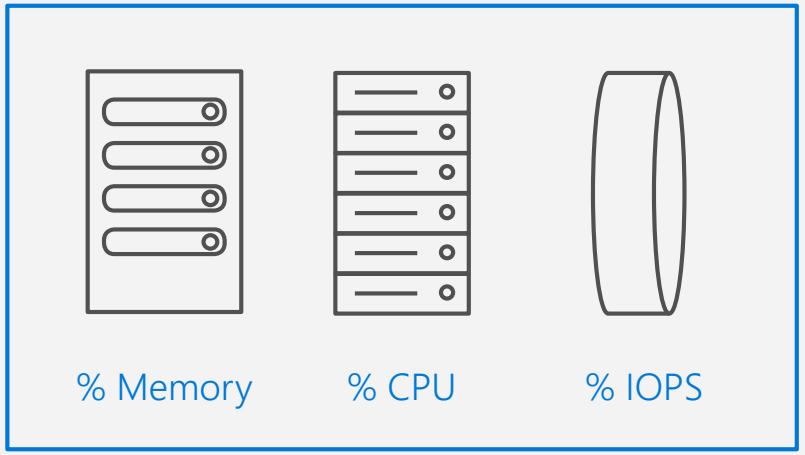
REQUEST UNITS

Request Units (RUs) is a rate-based currency

Abstracts physical resources for performing requests

Key to multi-tenancy, SLAs, and COGS efficiency

Foreground and background activities



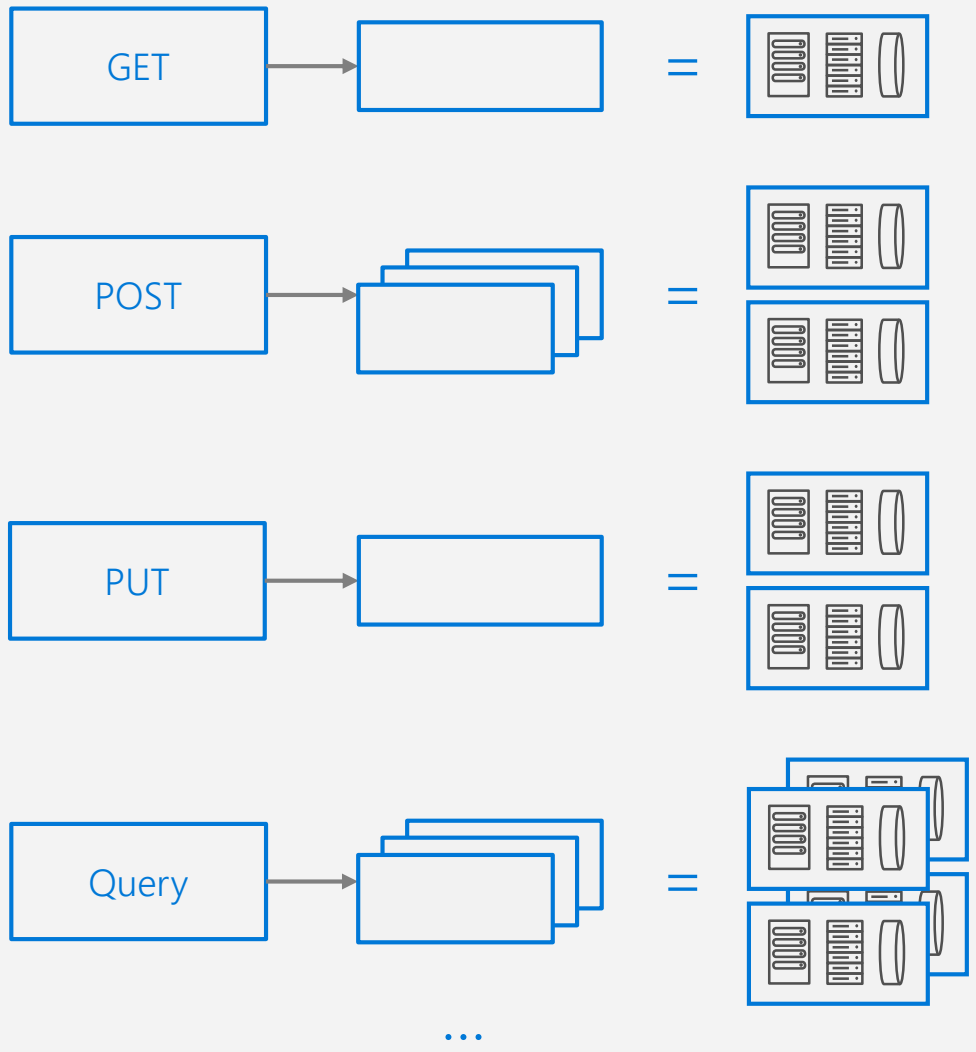
REQUEST UNITS

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



REQUEST UNITS

Normalized across various access methods

1 read of 1 KB document from a single partition

Each request consumes fixed RUs

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

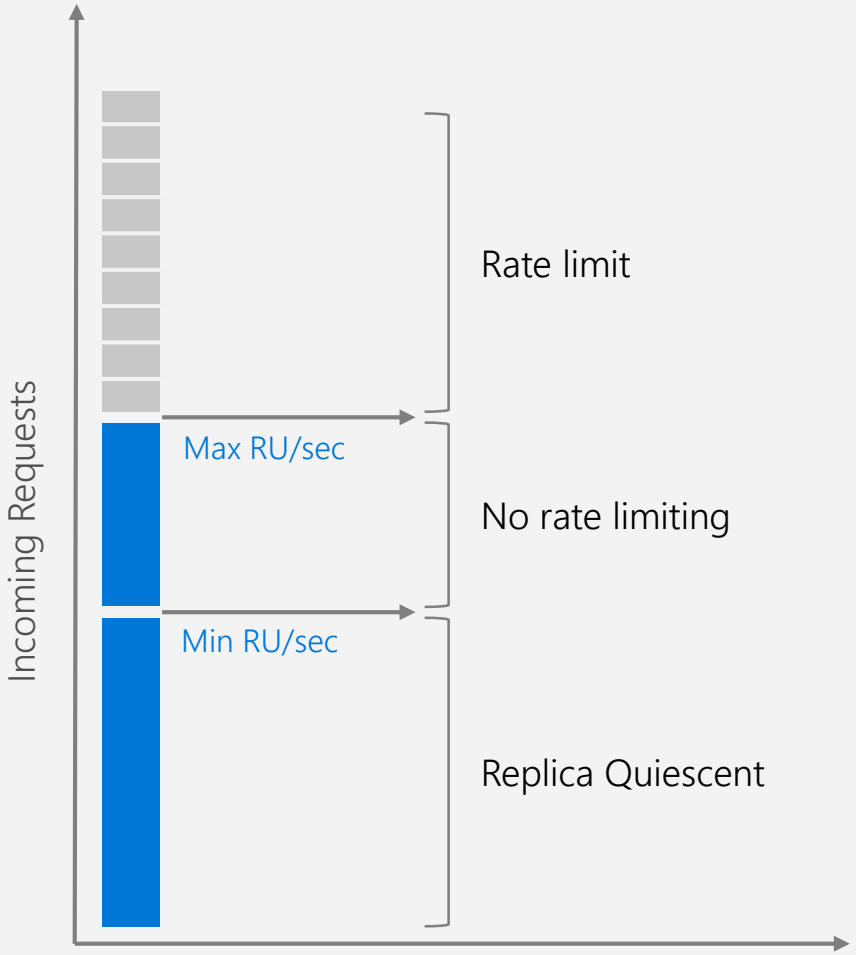
Provisioned in terms of RU/sec

Rate limiting based on amount of throughput provisioned

Can be increased or decreased instantaneously

Metered Hourly

Background processes like TTL expiration, index transformations scheduled when quiescent

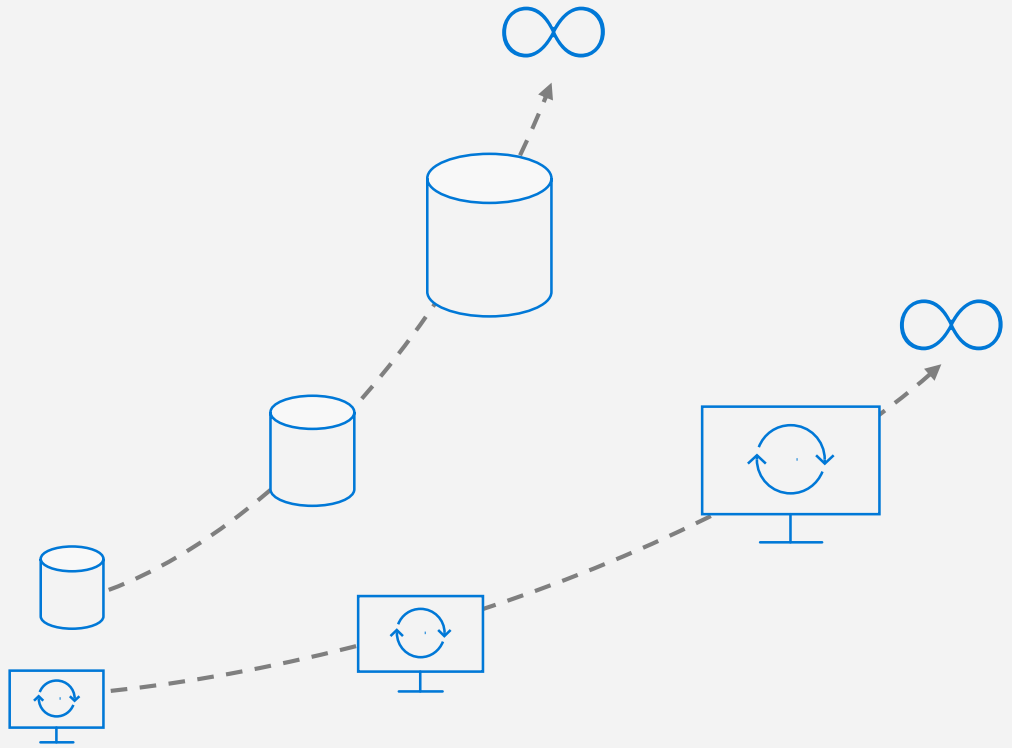


ELASTIC SCALE OUT OF STORAGE AND THROUGHPUT

SCALES AS YOUR APPS' NEEDS CHANGE

Independently and elastically scale storage and throughput across regions – even during unpredictable traffic bursts – with a database that adapts to your app's needs.

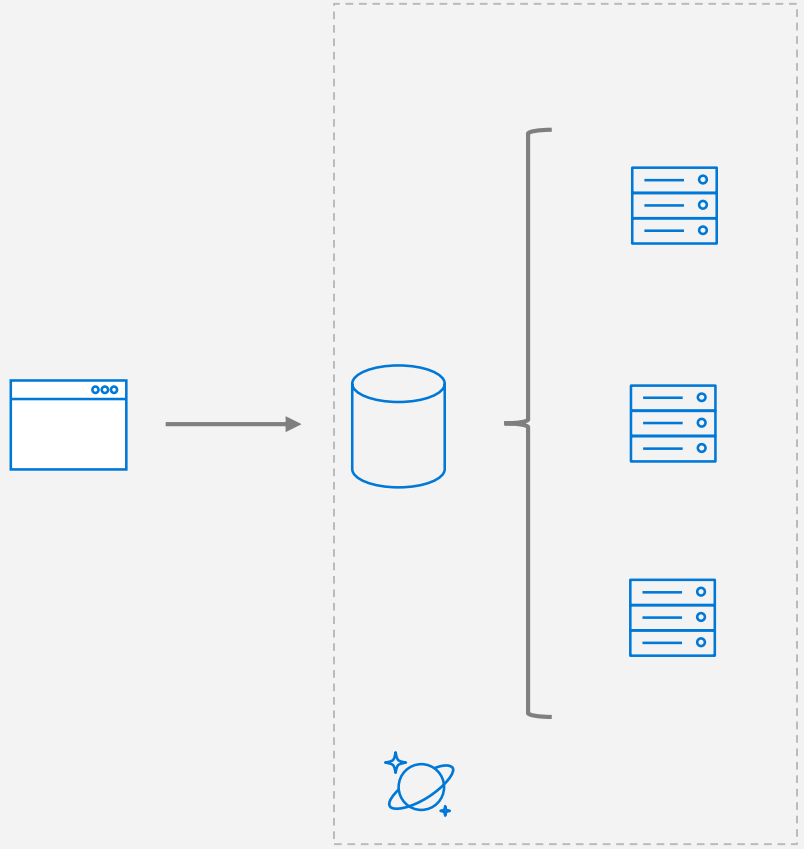
- Elastically scale throughput from 10 to 100s of millions of requests/sec across multiple regions
- Support for requests/sec for different workloads
- Pay only for the throughput and storage you need



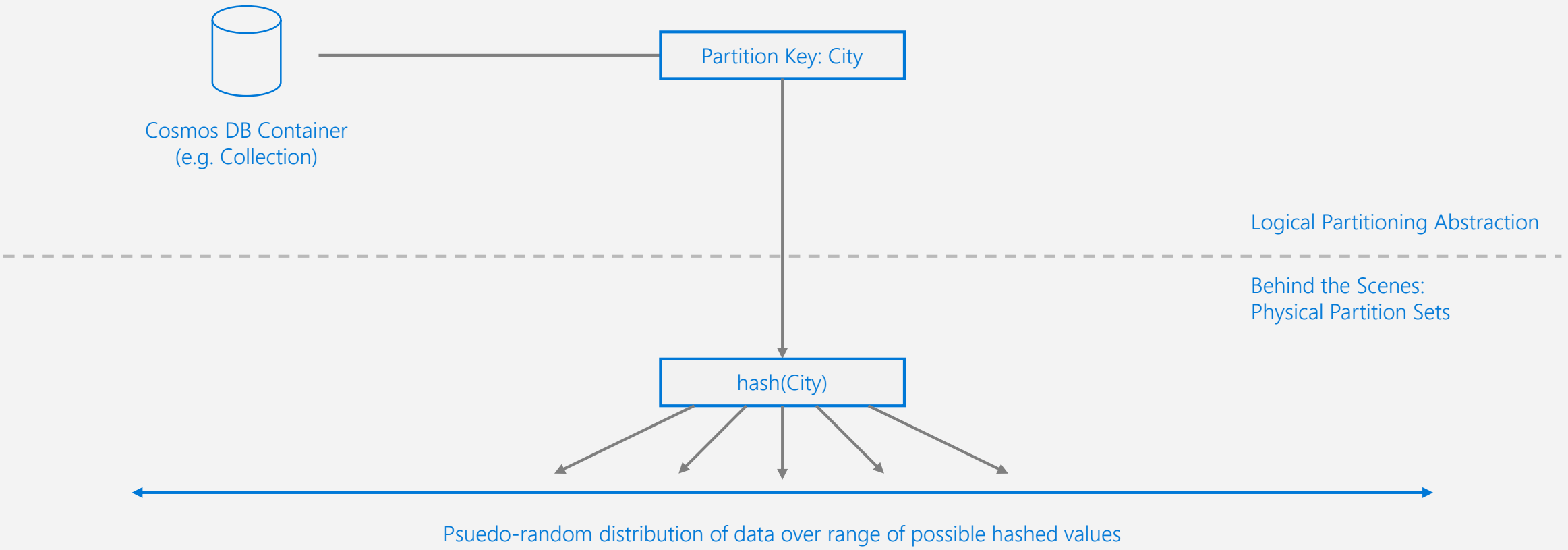
PARTITIONING

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

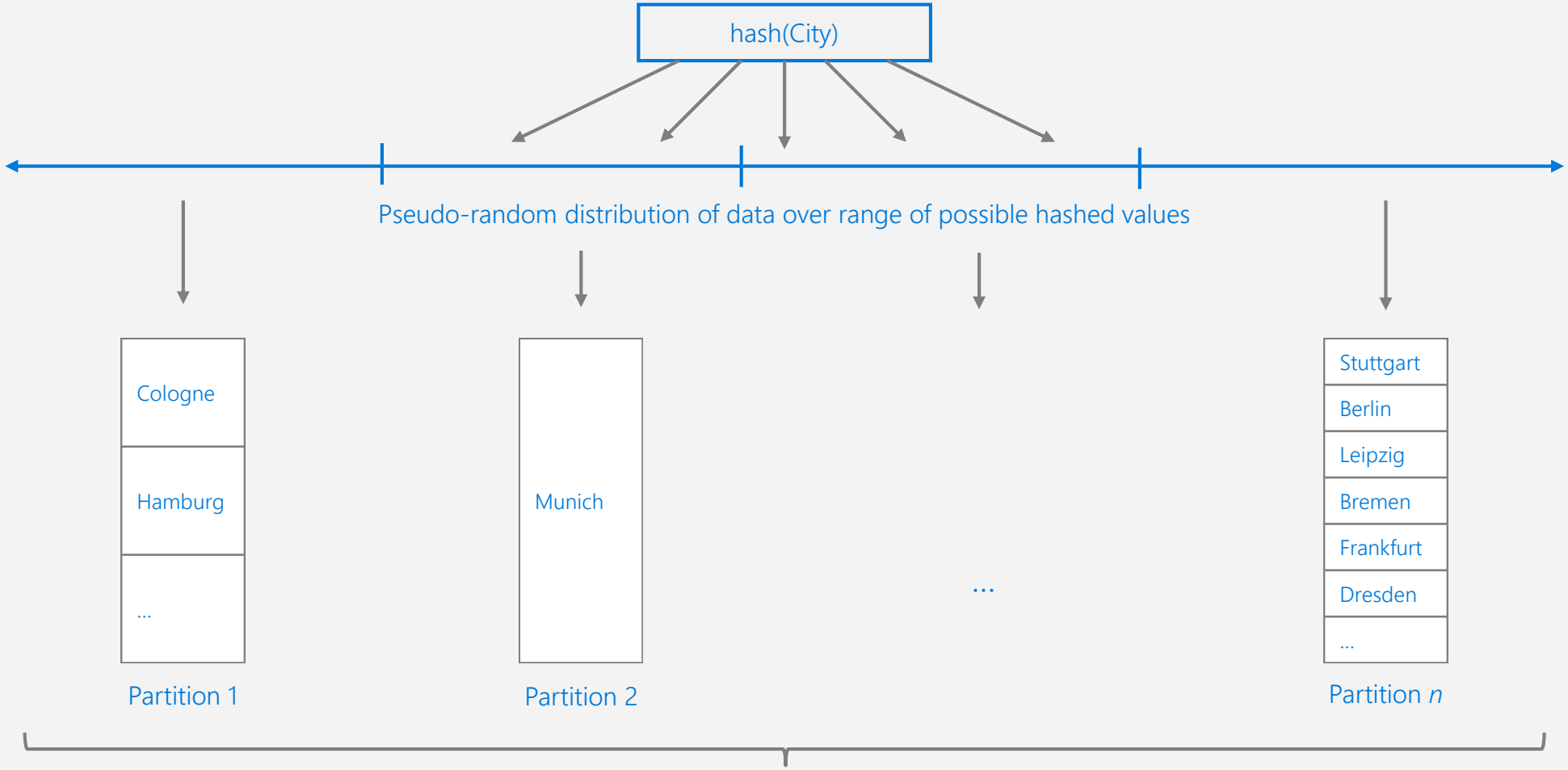
This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.



PARTITIONS

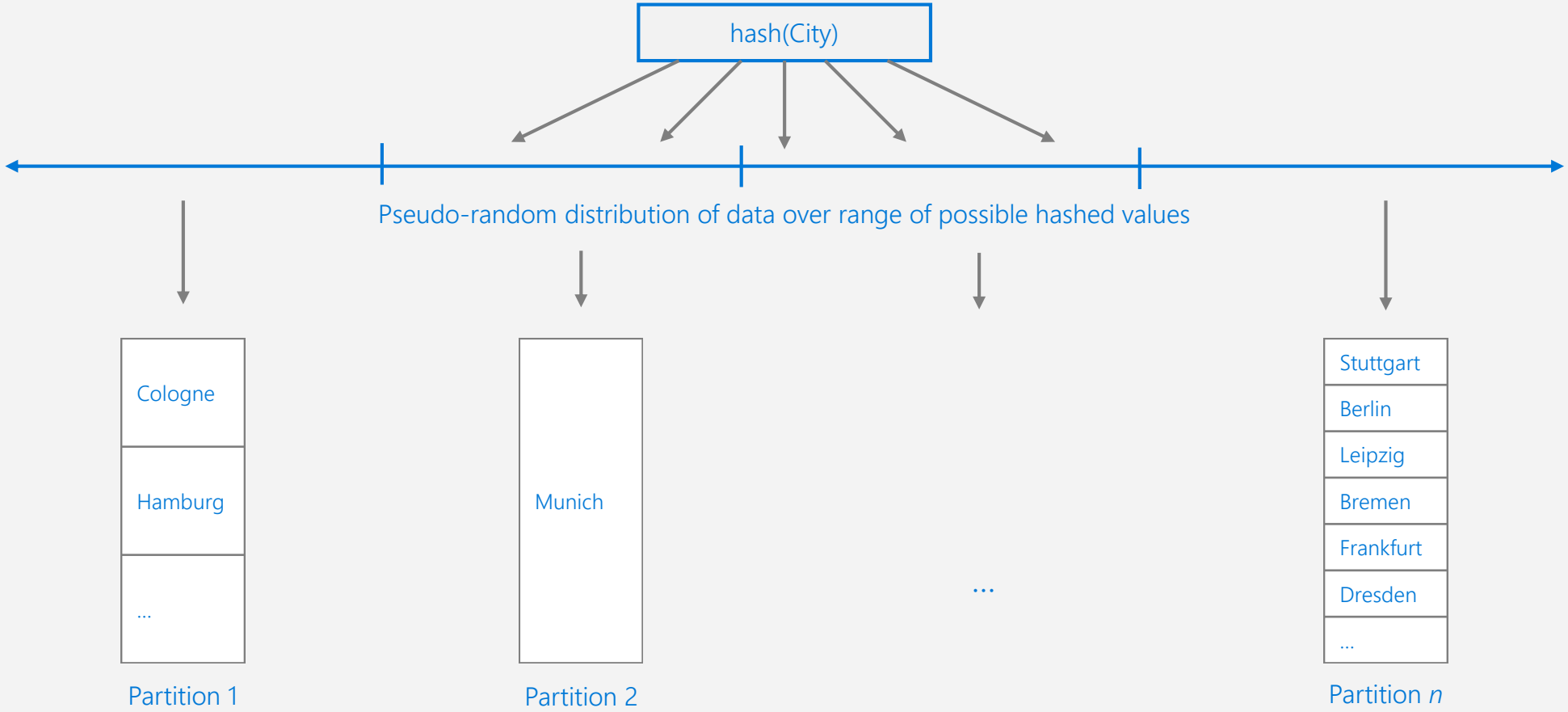

Partition Key: City
Cosmos DB Container
(e.g. Collection)
Logical Partitioning Abstraction
Behind the Scenes:
Physical Partition Sets
hash(City)
Psuedo-random distribution of data over range of possible hashed values

PARTITIONS



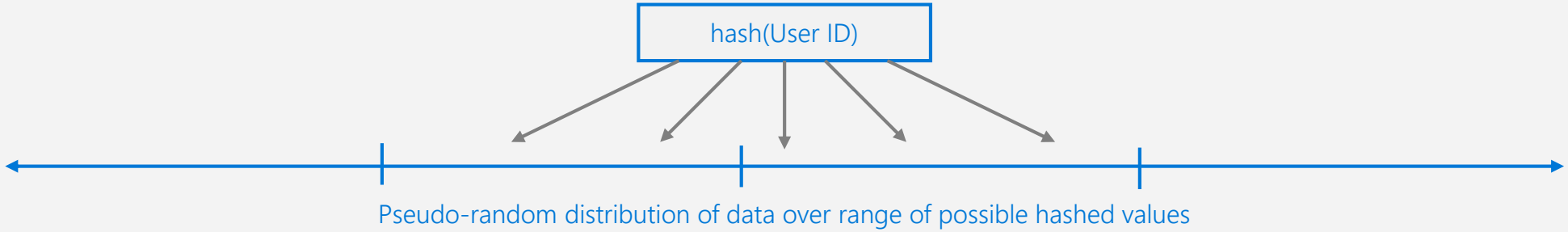
Frugal # of Partitions based on actual storage and throughput needs
(yielding scalability with low total cost of ownership)

PARTITIONS



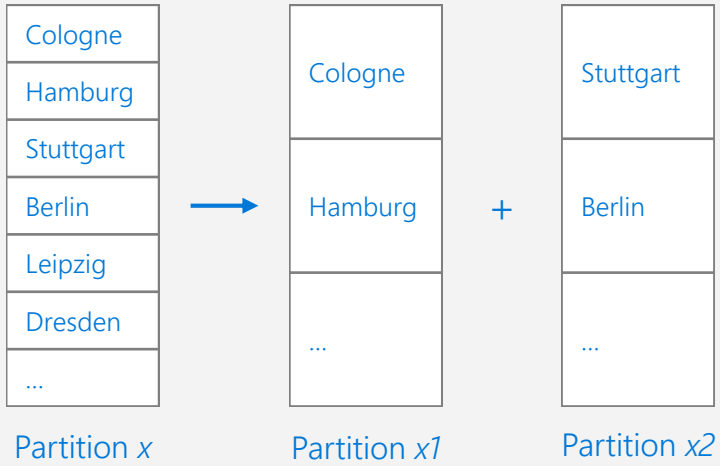
What happens when partitions need to grow?

PARTITIONS



Partition Ranges can be dynamically sub-divided to seamlessly grow database as the application grows while simultaneously maintaining high availability.

Partition management is fully managed by Azure Cosmos DB, so you don't have to write code or manage your partitions.



PARTITIONS

Best Practices: Design Goals for Choosing a Good Partition Key

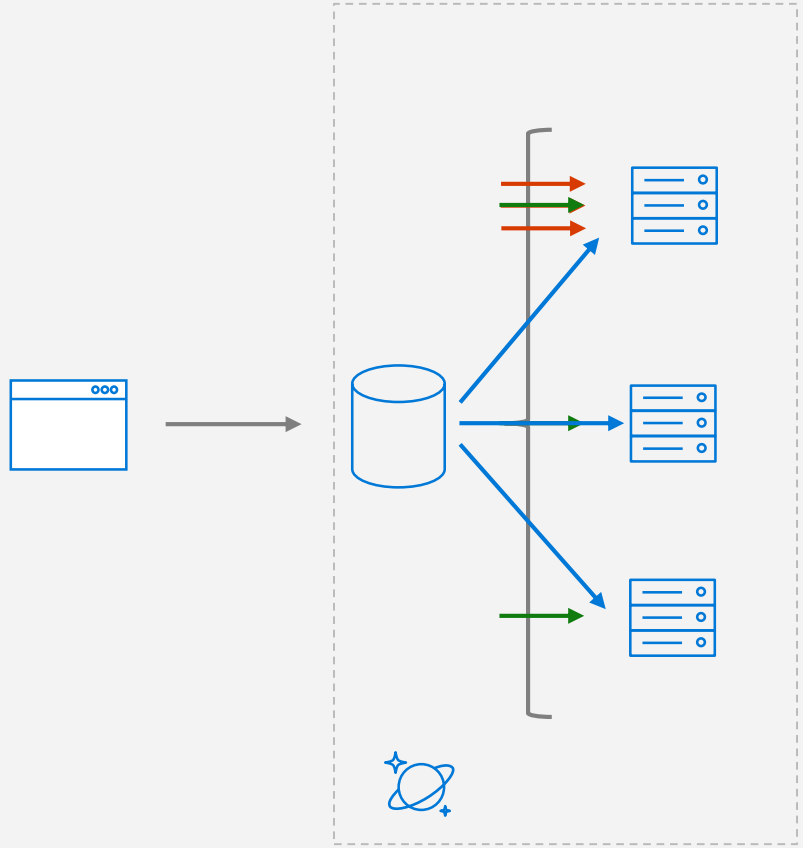
- Distribute the overall request + storage volume
  - Avoid "hot" partition keys
- Partition Key is scope for multi-record transactions and routing queries
  - Queries can be intelligently routed via partition key
  - Omitting partition key on query requires fan-out

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



DEMO

Selecting the partition key

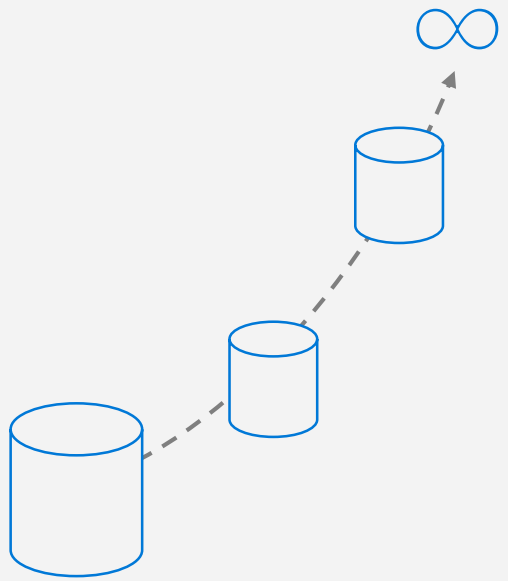
TURNKEY GLOBAL DISTRIBUTION

High Availability

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

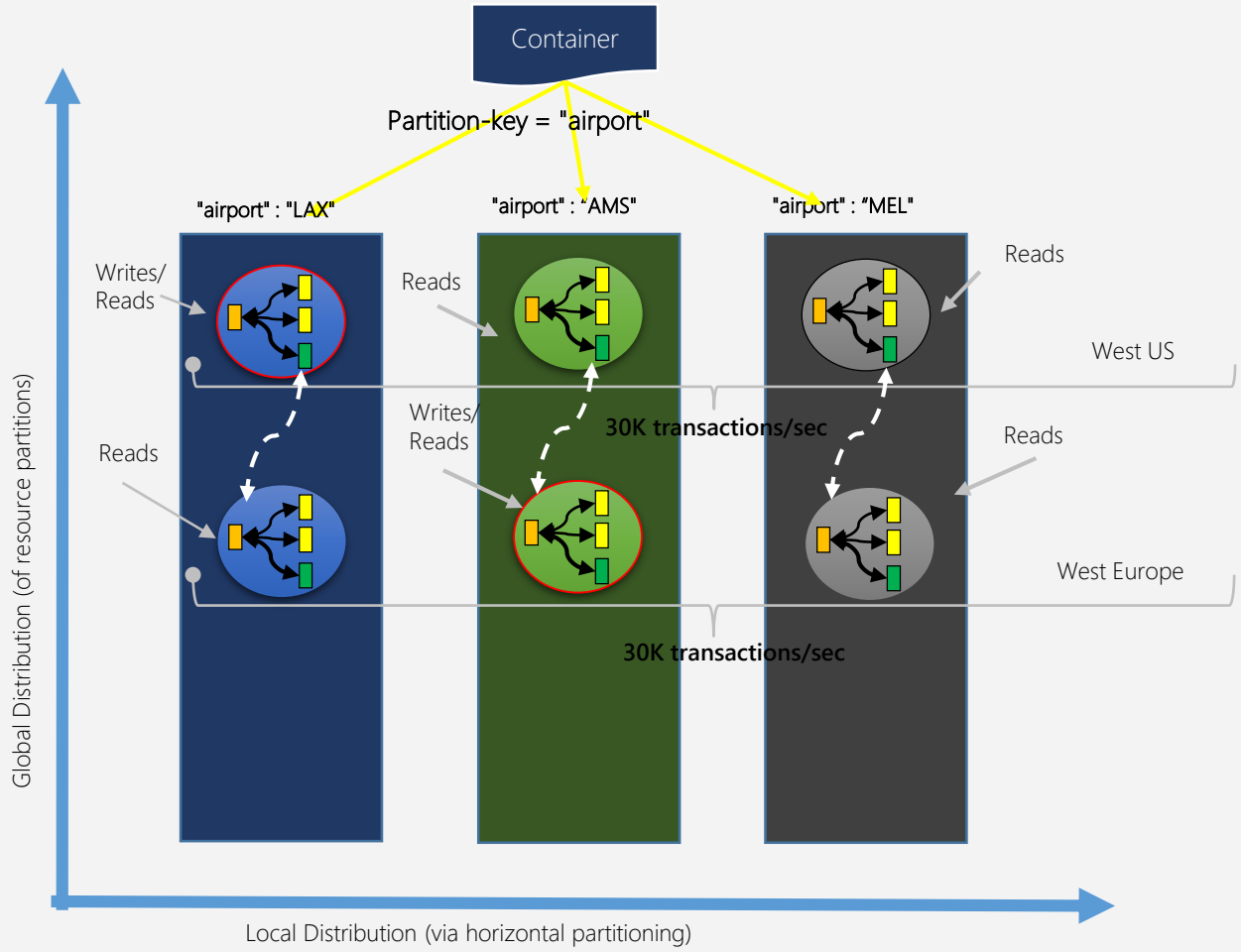
Low Latency (anywhere in the world)

- Packets cannot move fast 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

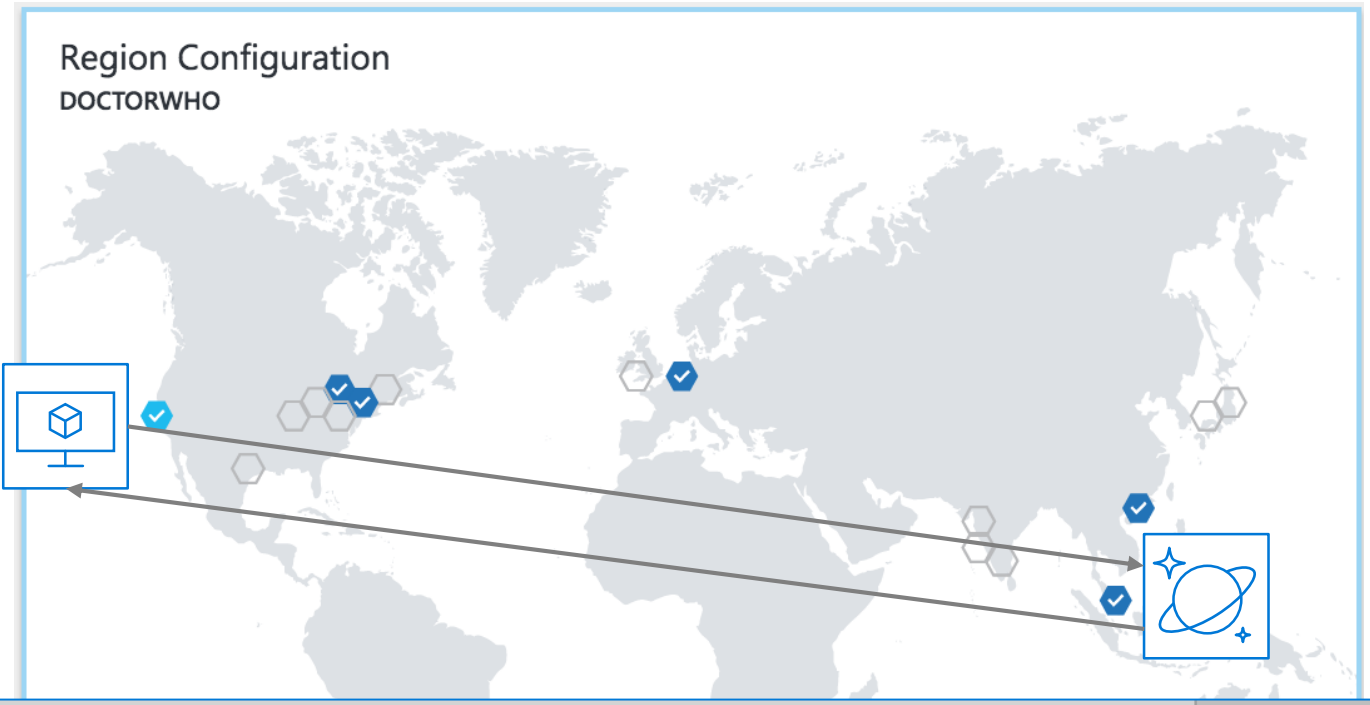


TURNKEY GLOBAL DISTRIBUTION

- 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



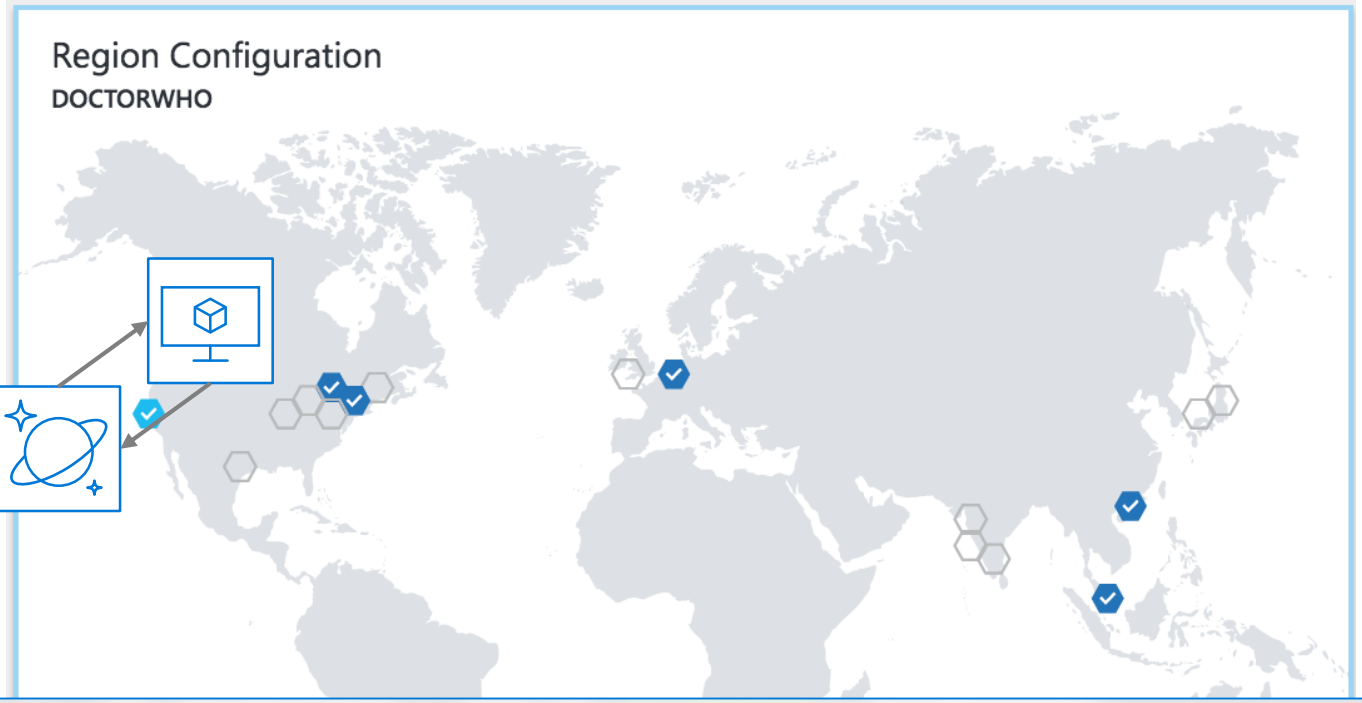
REPLICATING DATA GLOBALLY



...rkspace/docdb/docdb — amypond@blink: ~/docdb — ssh amypond@104.42.108.173 -p 22

```
amypond@blink:~/docdb$ node testQ2.js
210.788804 milliseconds, 2 RUs, ActivityId: 9025eac6-eb74-4a07-94cf-f2383caffbb3
178.825773 milliseconds, 2 RUs, ActivityId: ea53b736-b629-4290-9cdf-cf80c6139461
178.839173 milliseconds, 2 RUs, ActivityId: f143c992-ba67-4c7b-b7b8-0b5db8df4dbf
178.564573 milliseconds, 2 RUs, ActivityId: 1a8d7b5b-42c5-4c39-9160-d1e5ab2200d0
179.229073 milliseconds, 2 RUs, ActivityId: 483b85de-74e0-4f48-9206-70ac1268c60e
178.653772 milliseconds, 2 RUs, ActivityId: 50fbfe91-f41e-4f14-8a15-0b344c894727
178.464572 milliseconds, 2 RUs, ActivityId: cac6446a-79d4-4886-81d8-1dda835daa72
180.708475 milliseconds, 2 RUs, ActivityId: d40af8e4-582b-4479-bb9c-e3582eac6774
```

REPLICATING DATA GLOBALLY

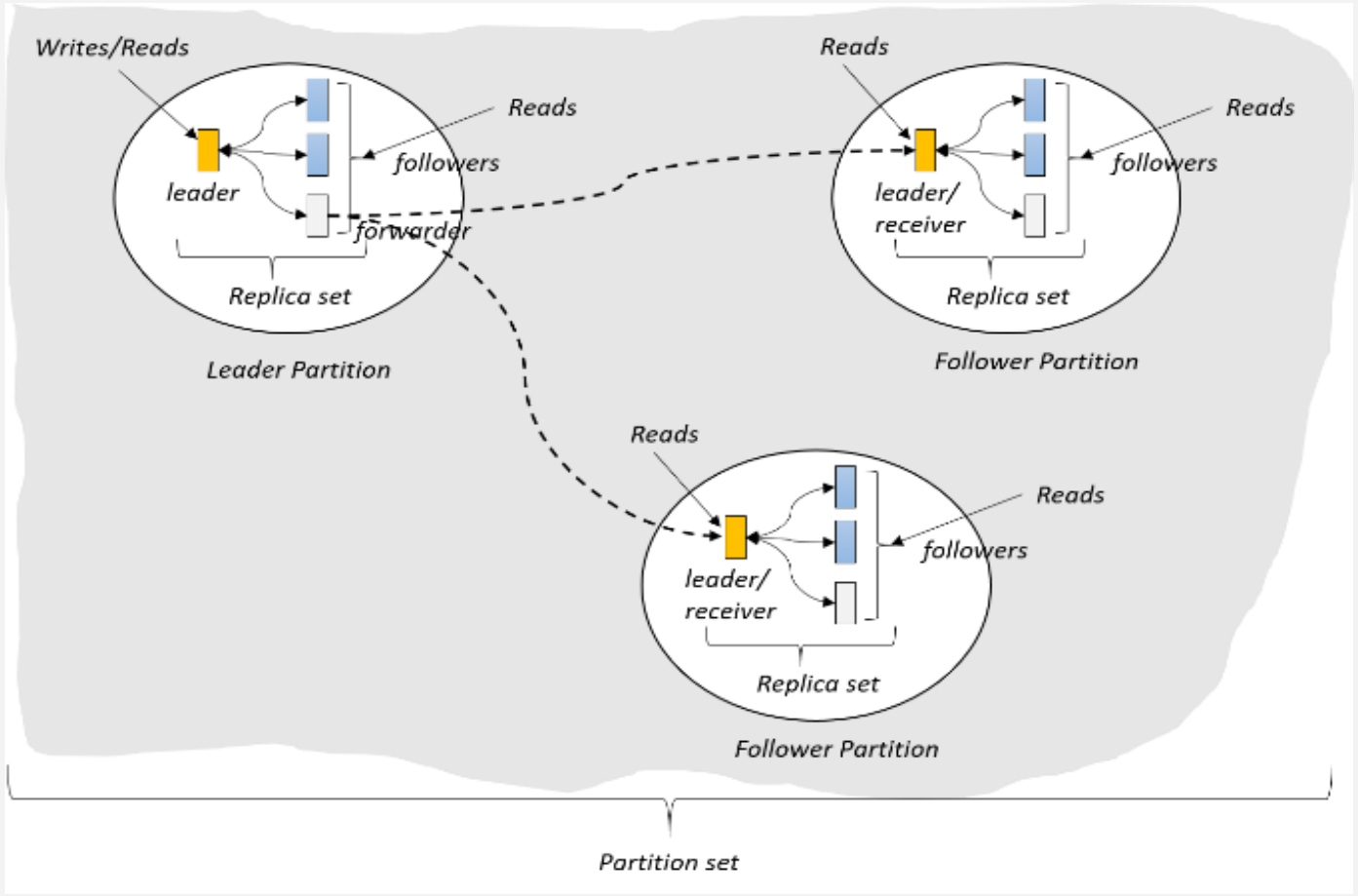



```
amypond@blink:~/docdb$ node testQ2.js
12.736112 milliseconds, 2 RUs, ActivityId: dd3c17b9-1b76-445a-8e27-29b7486bd7e4
4.947605 milliseconds, 2 RUs, ActivityId: e2f4c899-9fb1-4f76-a4ab-e5718fac5742
5.044005 milliseconds, 2 RUs, ActivityId: 0fc5d216-78a0-4d92-a3d0-63efd9dd6552
5.351205 milliseconds, 2 RUs, ActivityId: 861155f0-81ba-4c8a-9933-e50d8708cc21
4.553505 milliseconds, 2 RUs, ActivityId: 3db9641f-70f1-4ef1-84bb-809280bbe1a5
5.427506 milliseconds, 2 RUs, ActivityId: 10d1b2e5-e795-4c77-8655-815f410ba11e
5.900106 milliseconds, 2 RUs, ActivityId: bda1df86-c5ad-45c5-bcfb-93d417c54751
4.895405 milliseconds, 2 RUs, ActivityId: f206d58d-64a2-47f2-9653-145eaf47ff97
5.244306 milliseconds, 2 RUs, ActivityId: 3aa7e177-b1a9-413d-8023-55f5054d1b74
```

DEMO

Configure Global Distribution

BREWER'S CAP THEOREM



Impossible for distributed data store to simultaneously provide more than 2 out of the following 3 guarantees:

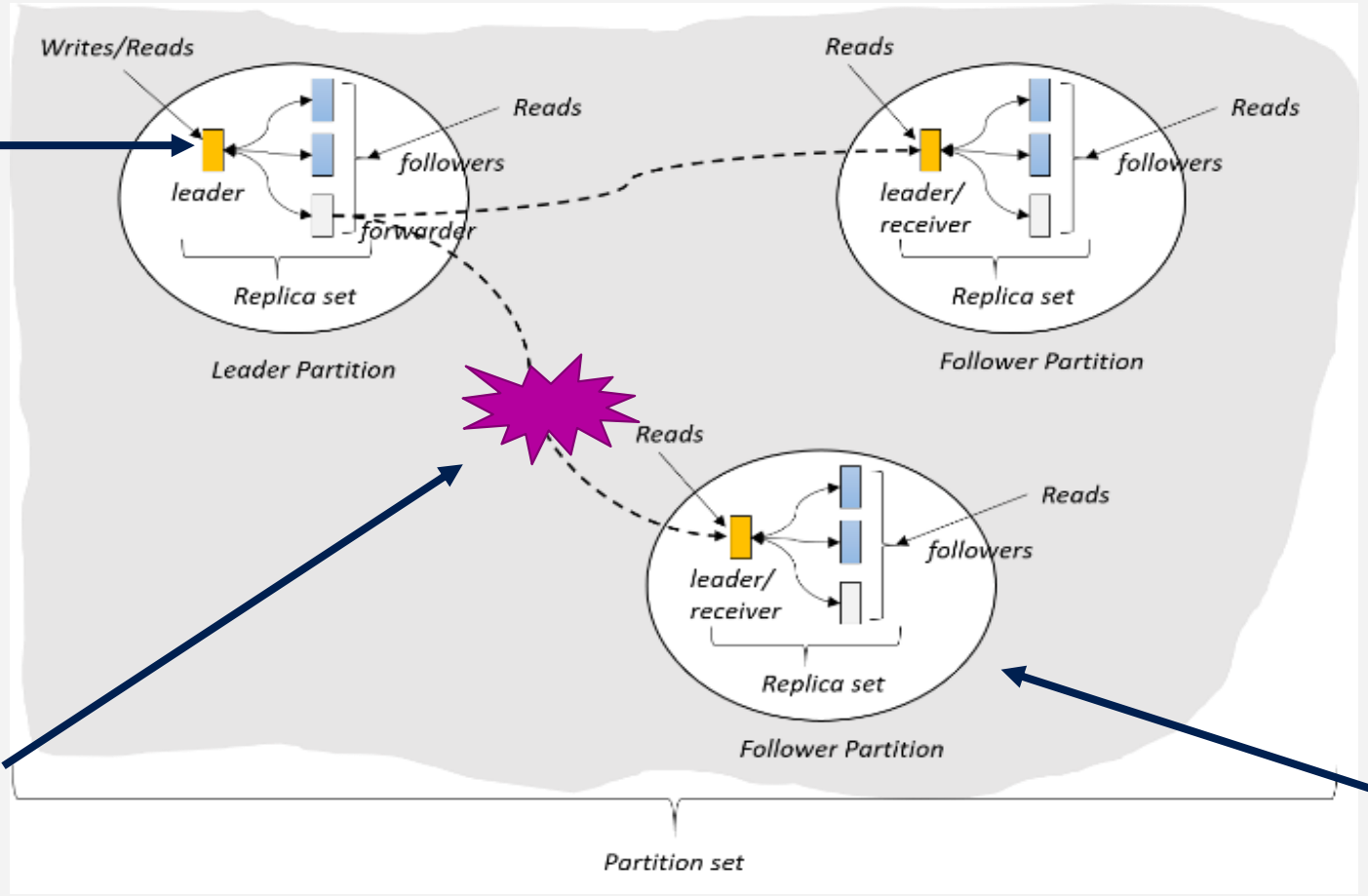
- Consistency
- Availability
- Partition Tolerance

CONSISTENCY

(West US)

Value = ~~5~~ 6

Update 5 => 6



(East US)

Value = ~~5~~ 6

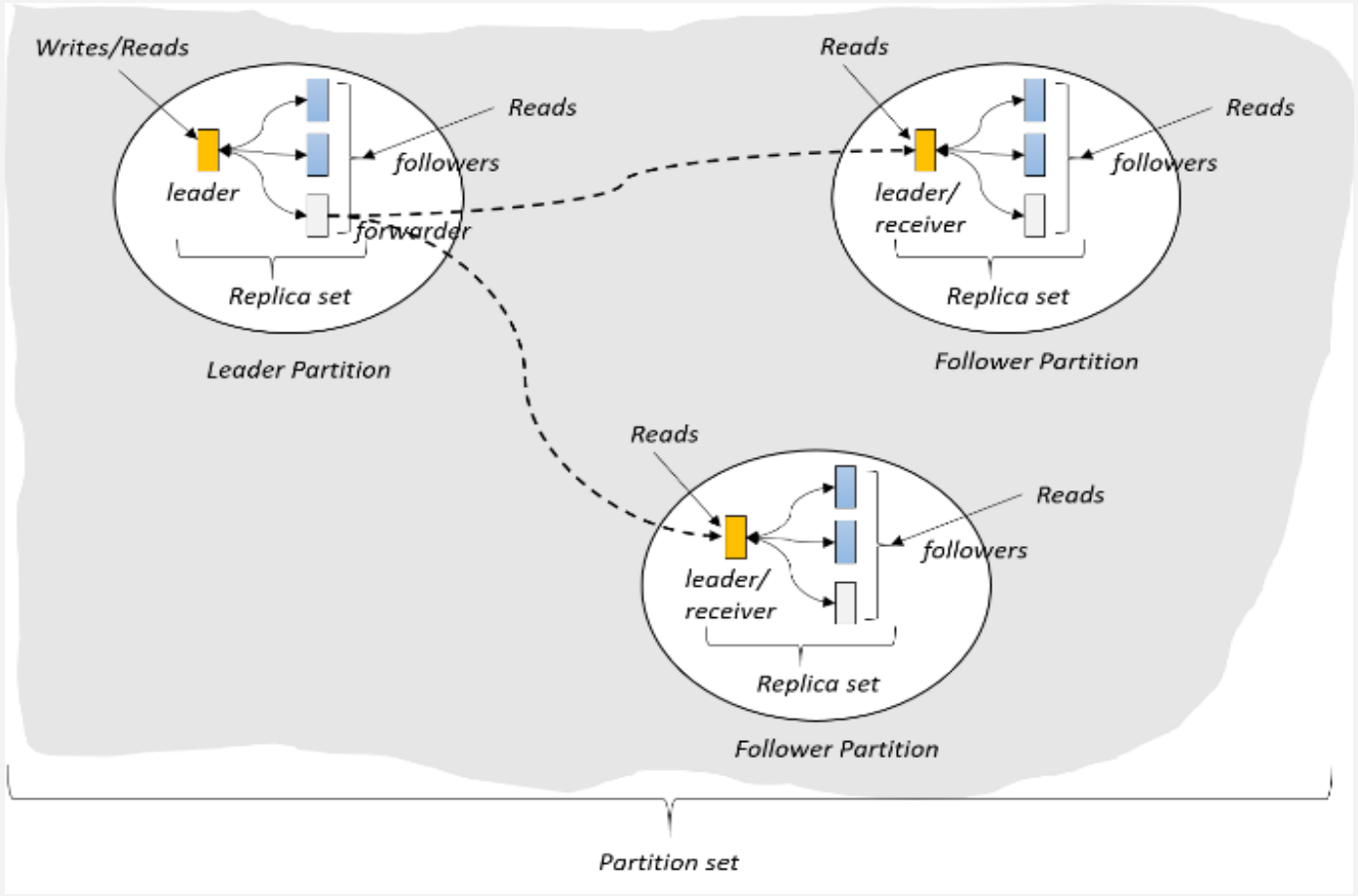
(North Europe)

Value = 5

What happens when a network partition is introduced?

Reader: What is the value? Should it see 5? (prioritize availability) Or does the system go offline until network is restored? (prioritize consistency)

PACELC THEOREM



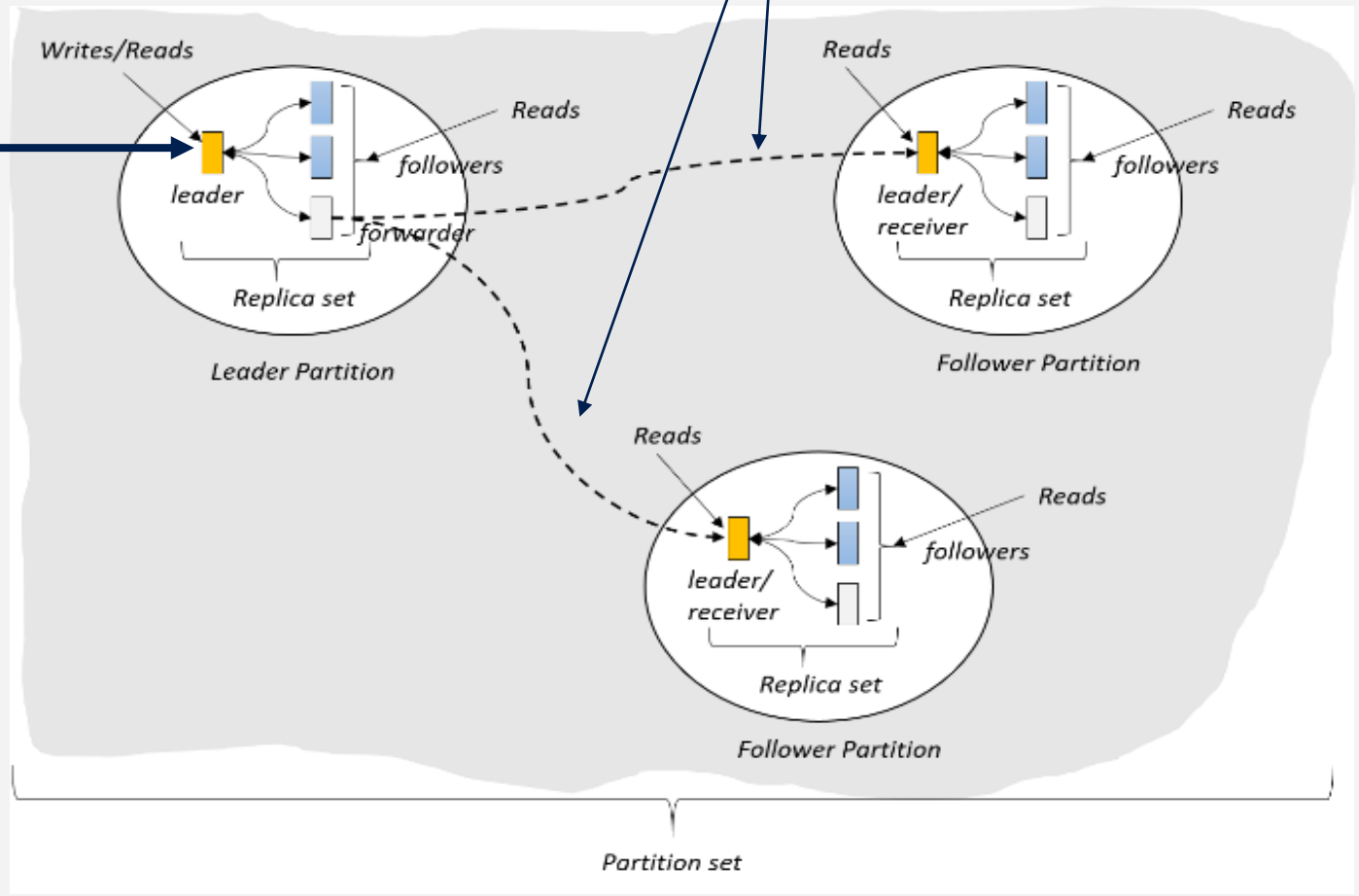
In the case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

CONSISTENCY

Latency: packet of information can travel as fast as speed of light. Replication between distant geographic regions can take 100's of milliseconds

Value = ~~5~~ 6

Update 5 => 6



Value = ~~5~~ 6

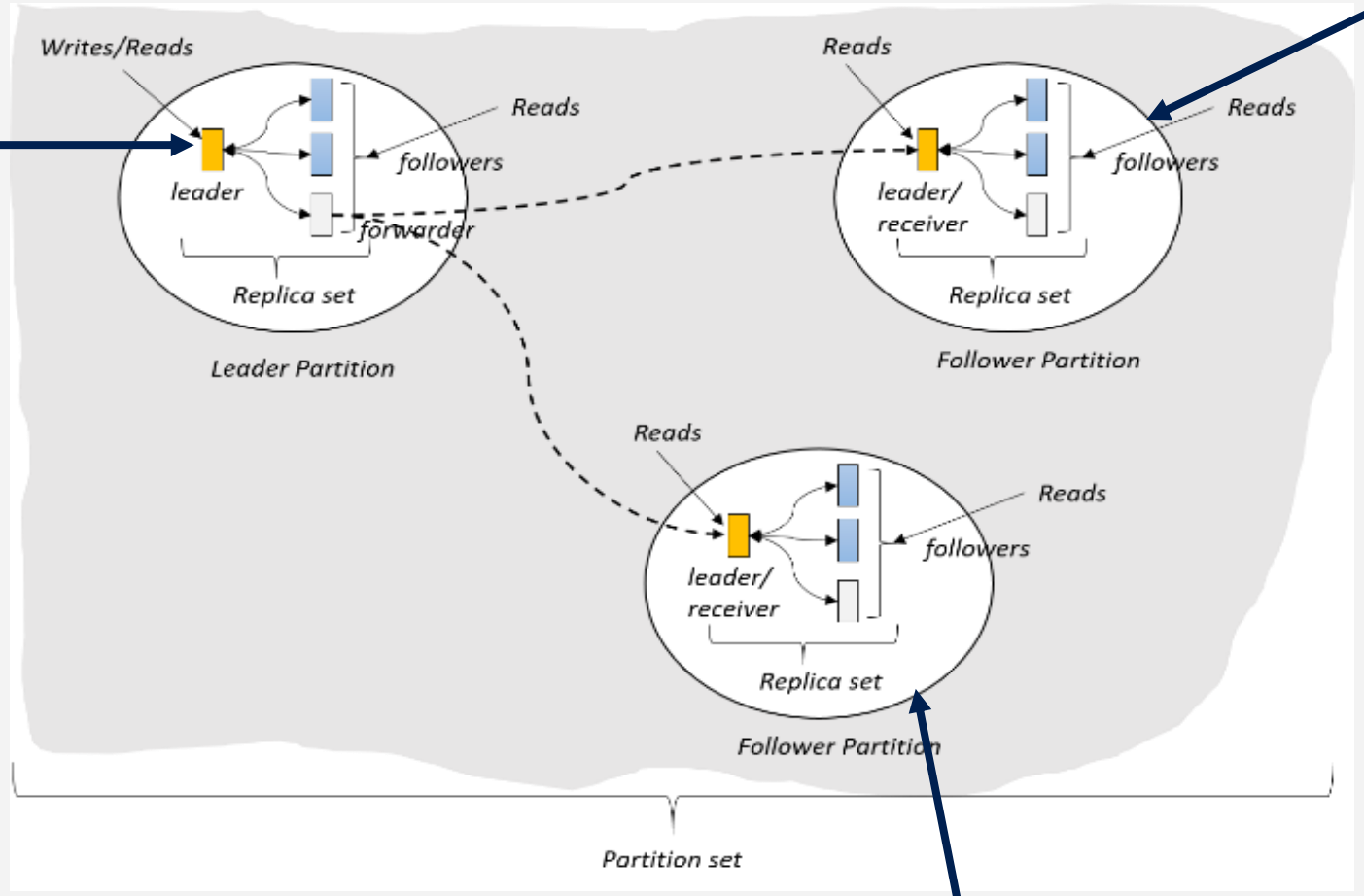
Value = 5

CONSISTENCY

Reader A: What is the value?

Value = ~~5~~ 6

Update 5 => 6



Value = ~~5~~ 6

Value = 5

Should it Reader B see 5 immediately? (prioritize latency)

Does it see the same result as reader A? (quorum impacts throughput)

Does it sit and wait for 5 => 6 propagate? (prioritize consistency)

Reader B: What is the value?

FIVE WELL-DEFINED CONSISTENCY MODELS

CHOOSE THE BEST CONSISTENCY MODEL FOR YOUR APP

Five well-defined, consistency models

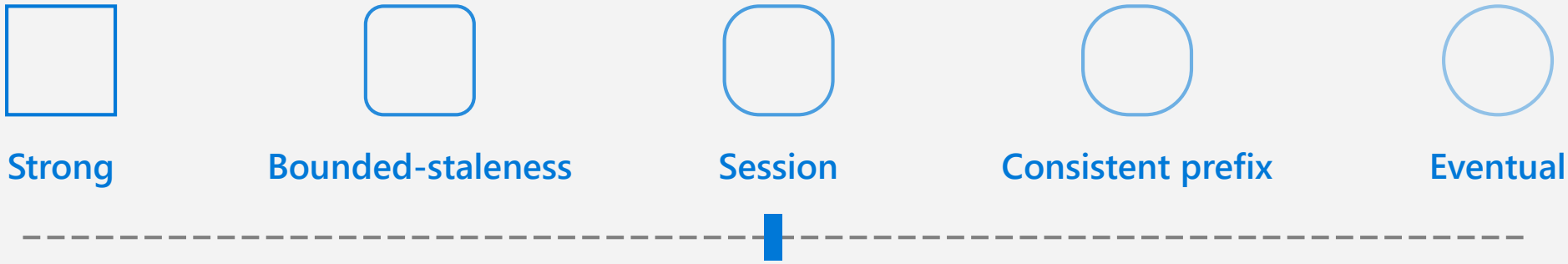
Overridable on a per-request basis

Provides control over performance-consistency tradeoffs, backed by comprehensive SLAs.

An intuitive programming model offering low latency and high availability for your planet-scale app.

CLEAR TRADEOFFS

- **Latency**
- **Availability**
- **Throughput**



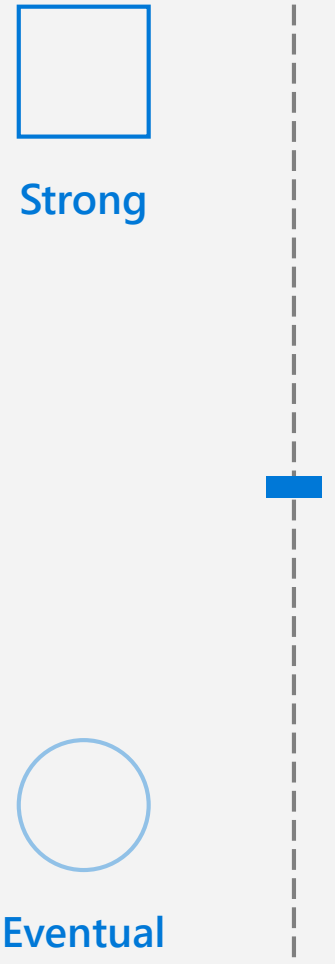
DEMYSTIFYING CONSISTENCY MODELS

Strong consistency

Guarantees linearizability. Once an operation is complete, it will be visible to all readers in a strongly-consistent manner across replicas.

Eventual consistency

Replicas are eventually consistent with any operations. There is a potential for out-of-order reads. Lowest cost and highest performance for reads of all consistency levels.



DEMYSTIFYING CONSISTENCY MODELS

Bounded-staleness

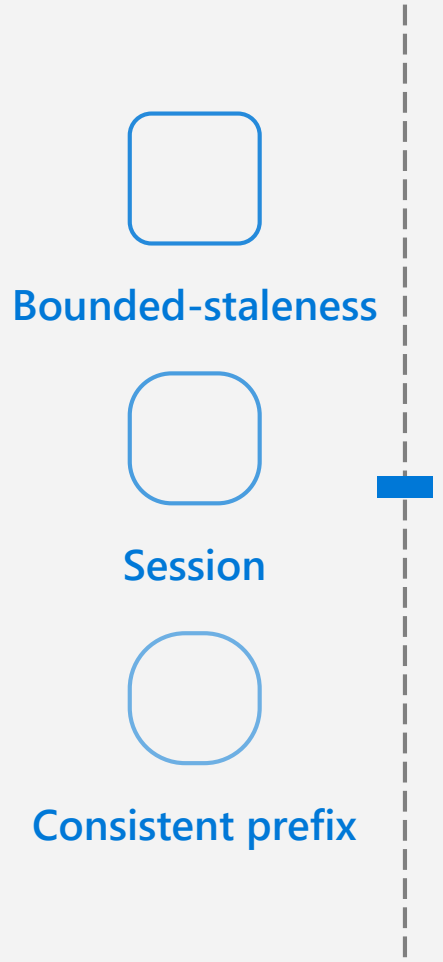
Consistent prefix. Reads lag behind writes by at most **k** prefixes or **t** interval. Similar properties to strong consistency except within staleness window.

Session

Consistent prefix. Within a session, reads and writes are monotonic. This is referred to as “read-your-writes” and “write-follows-reads”. Predictable consistency for a session. High read throughput and low latency outside of session.

Consistent Prefix

Reads will never see out of order writes.



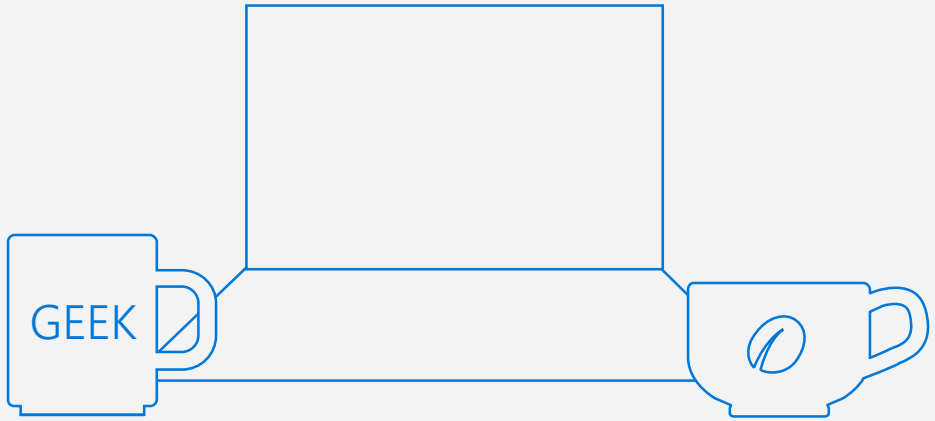
DEMO

Configure Consistency

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.

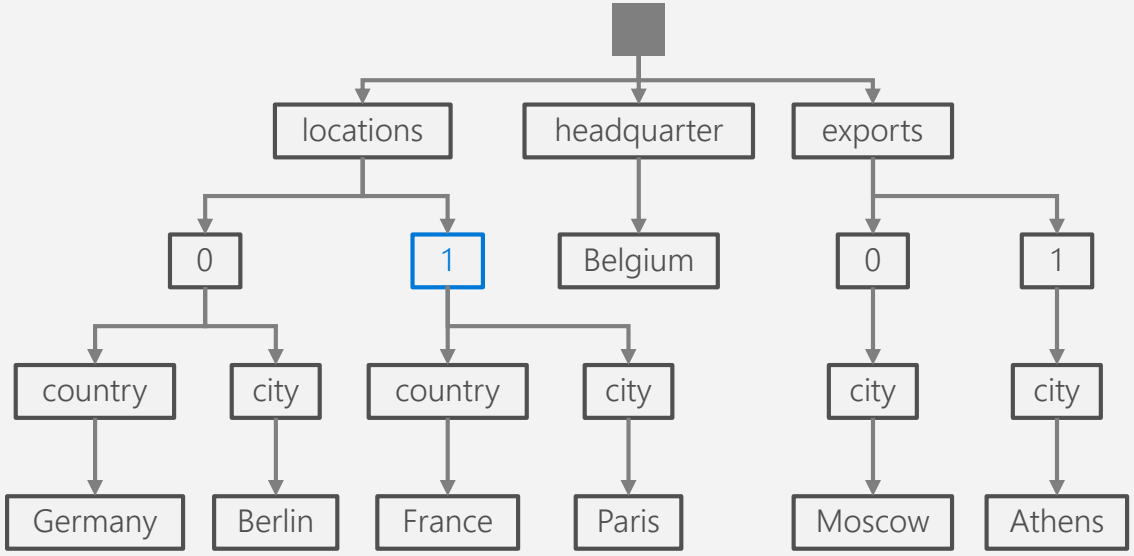
- Automatic index management
- Synchronous auto-indexing
- No schemas or secondary indices needed
- Works across every data model



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

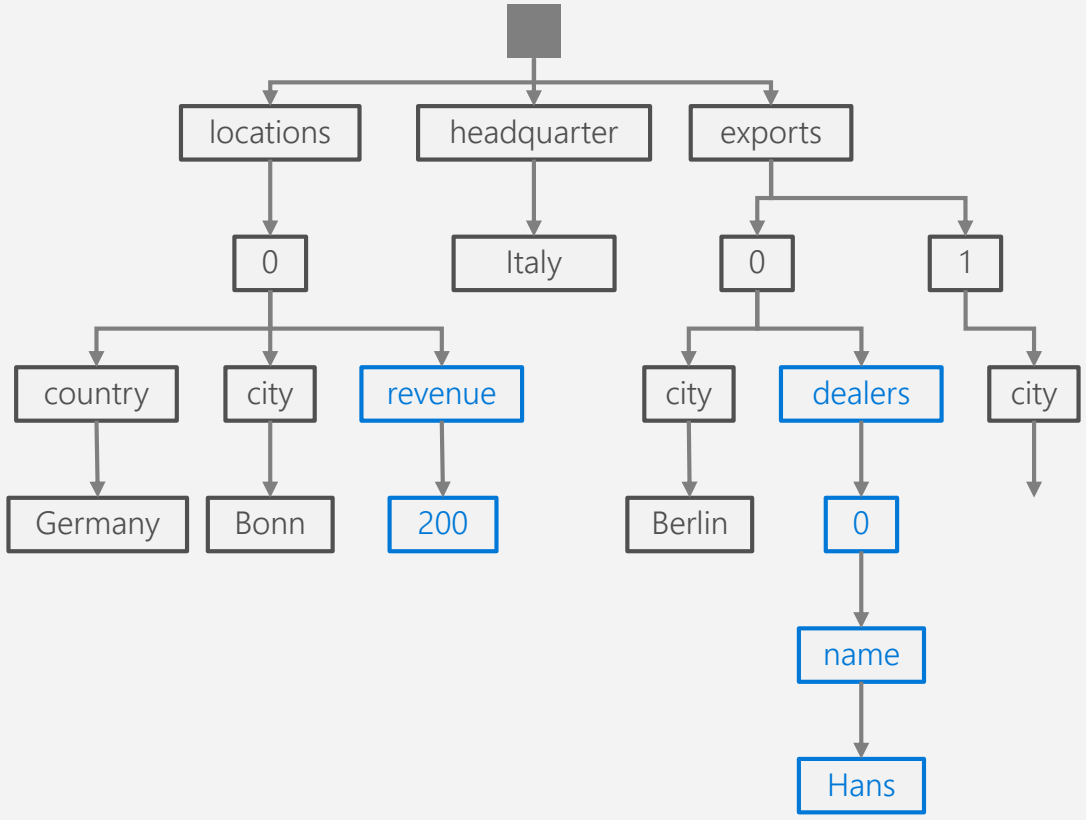
INDEXING JSON DOCUMENTS

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

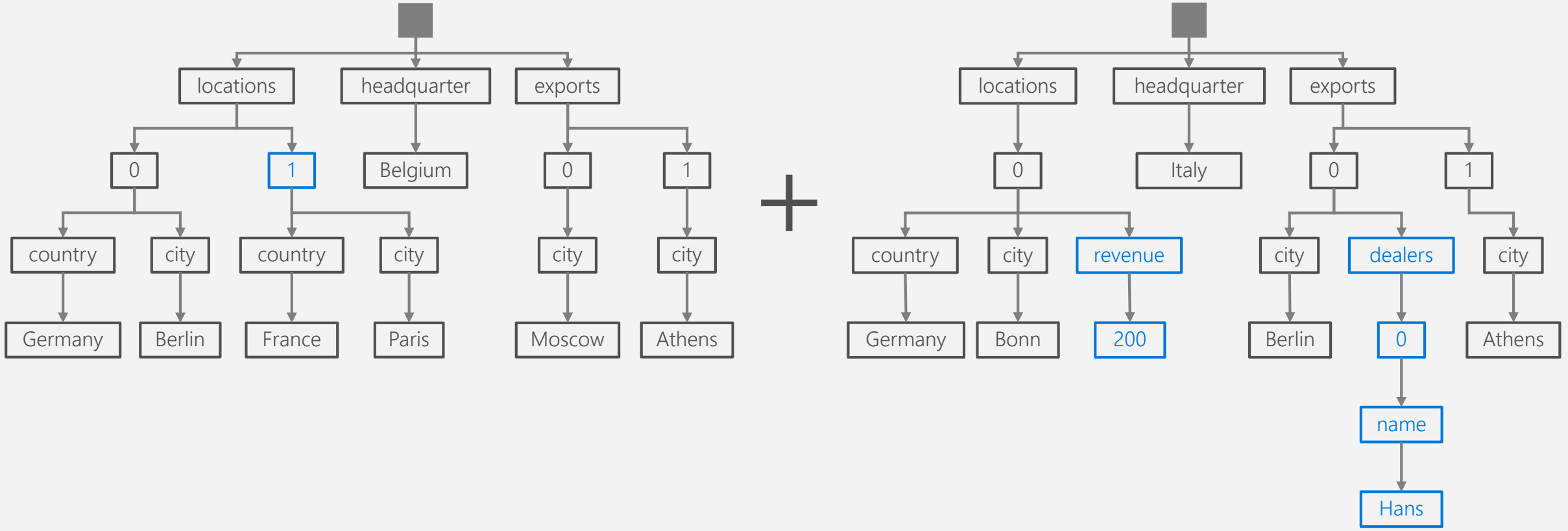


INDEXING JSON DOCUMENTS

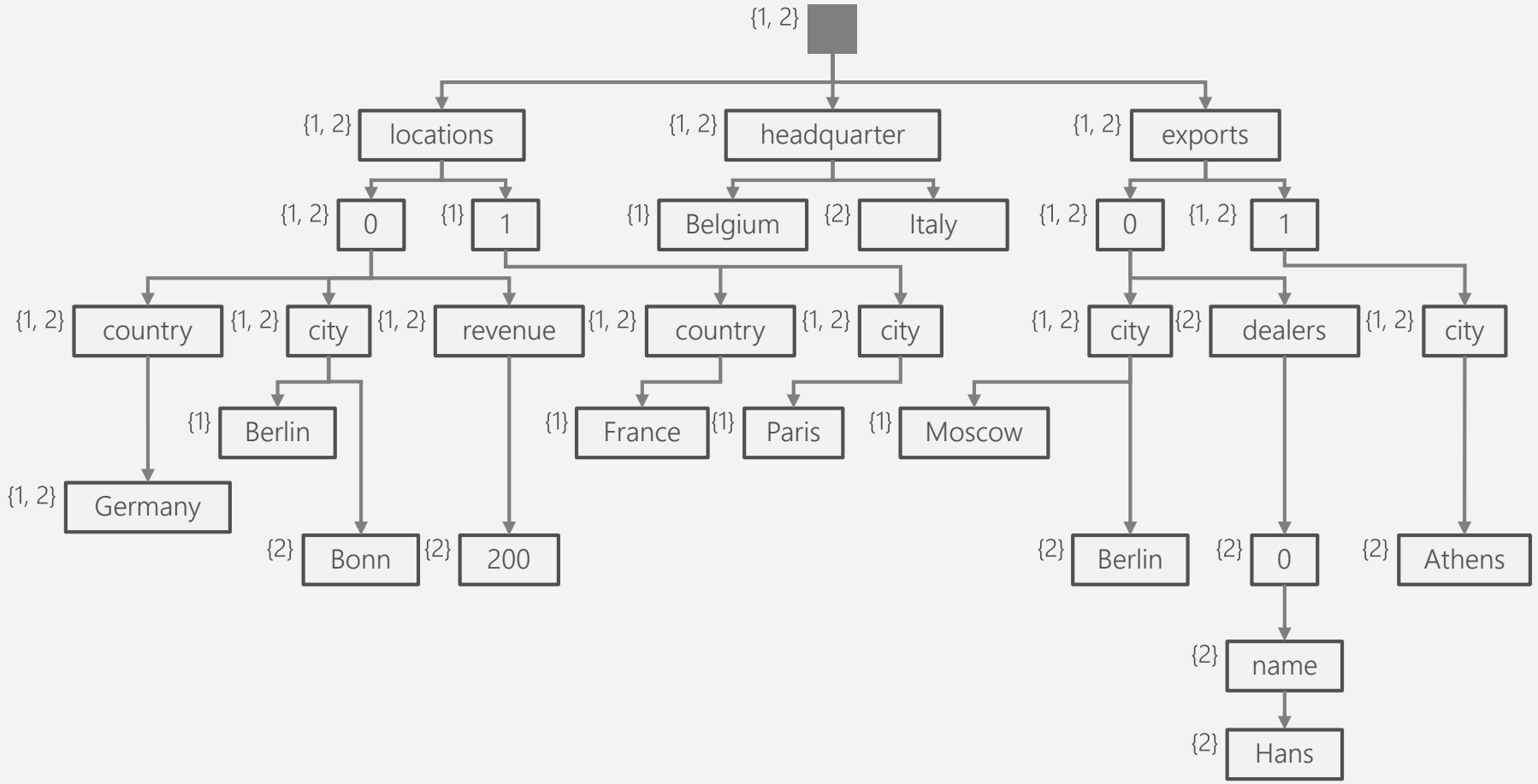
```
{
    "locations": [
        {
            "country": "Germany",
            "city": "Bonn",
            "revenue": 200
        }
    ],
    "headquarter": "Italy",
    "exports": [
        {
            "city": "Berlin",
            "dealers": [
                { "name": "Hans" }
            ]
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



INDEXING JSON DOCUMENTS



INVERTED INDEX



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/*"
    }]
}
```

DEMO

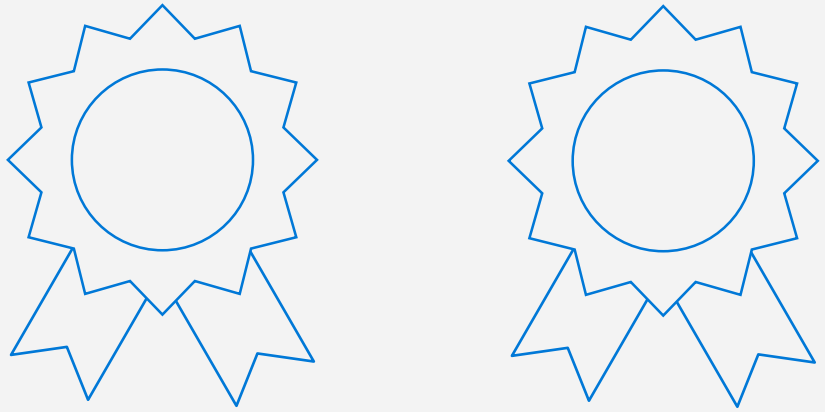
Indexing Policy

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)

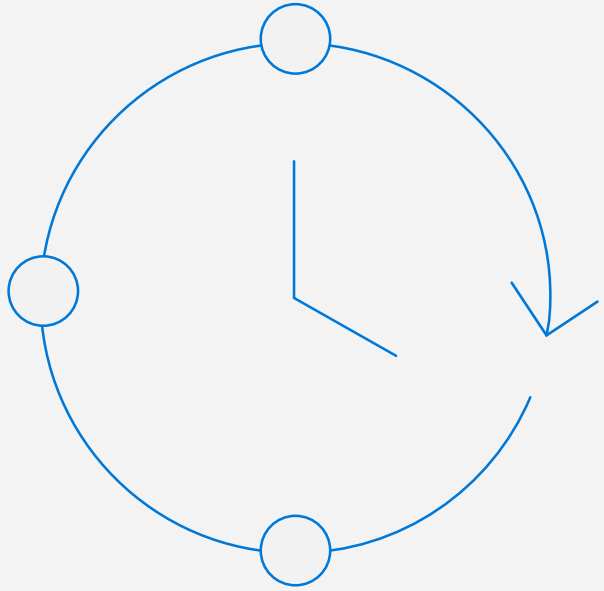
AUTOMATICALLY PURGE DATA

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.



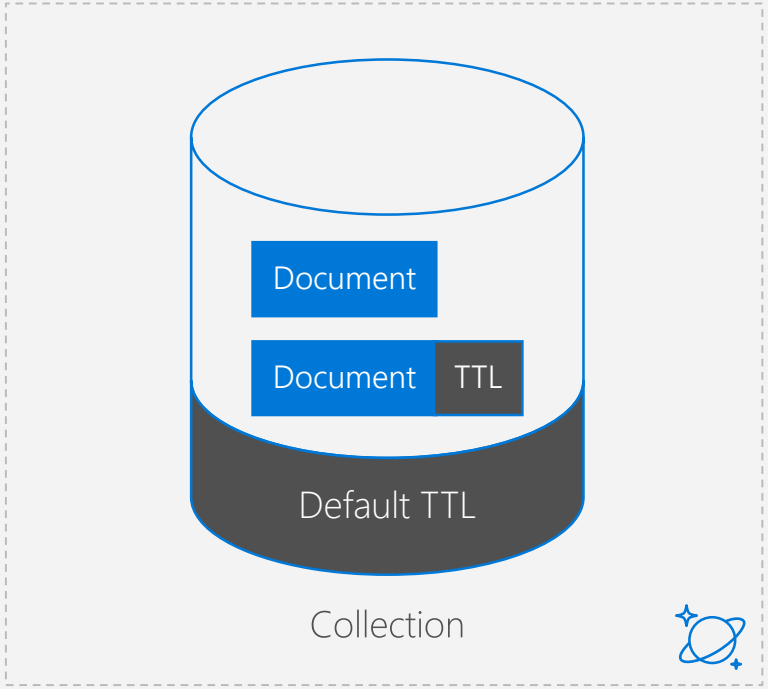
EXPIRING RECORDS USING TIME-TO-LIVE

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.



DEMO

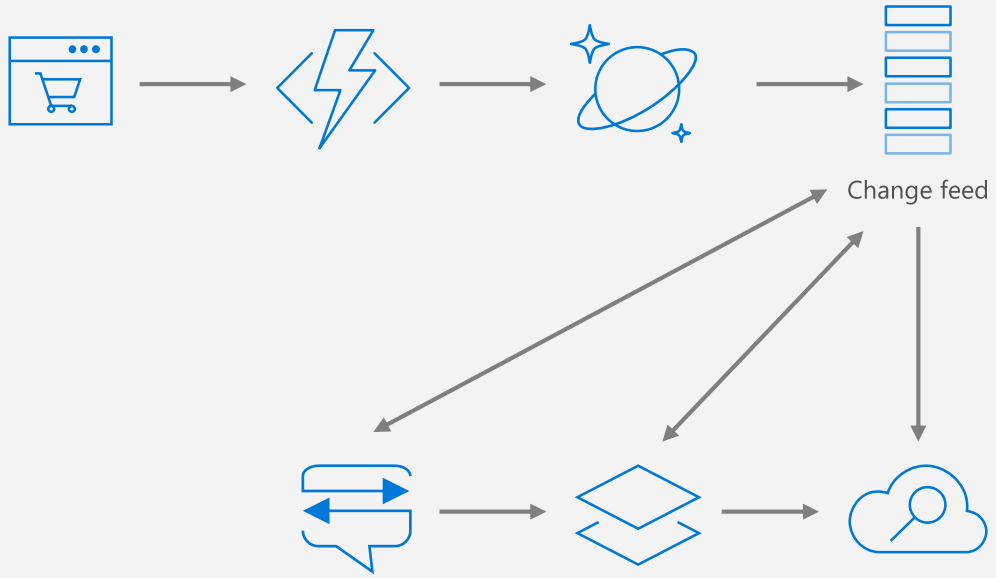
Pruning old records using TTL Values

MODERN REACTIVE APPLICATIONS

IoT, gaming, retail and operational logging applications need to **track and respond to tremendous amount of data** being ingested, modified or removed from a globally-scaled database.

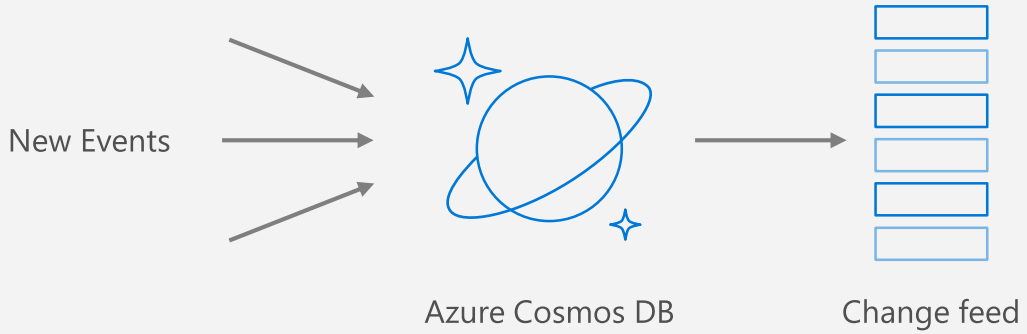
COMMON SCENARIOS

- Trigger notification for new items
- Perform real-time analytics on streamed data
- Synchronize data with a cache, search engine or data warehouse.

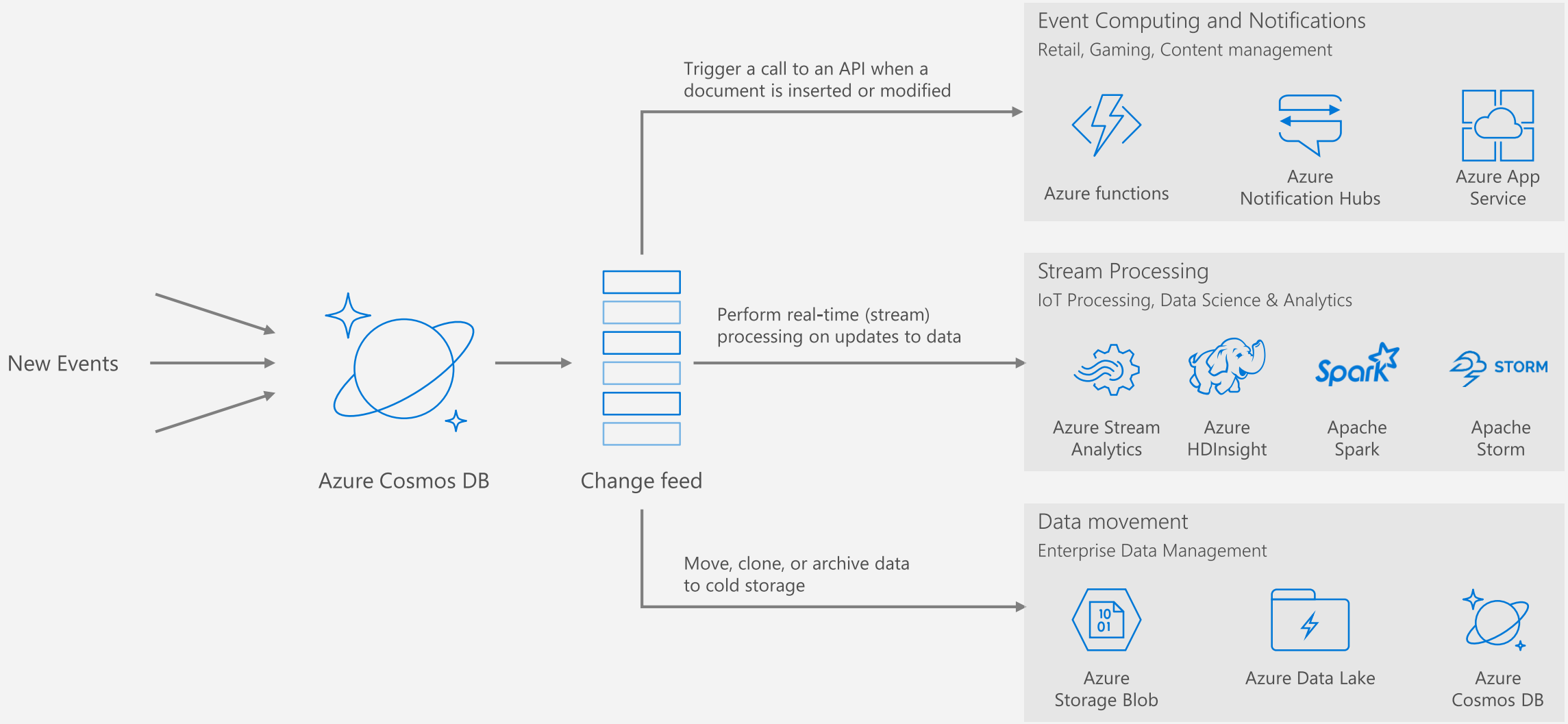


CHANGE FEED

Persistent log of records within an Azure Cosmos DB container. Preseneted in the order in which they were modified



CHANGE FEED SCENARIOS

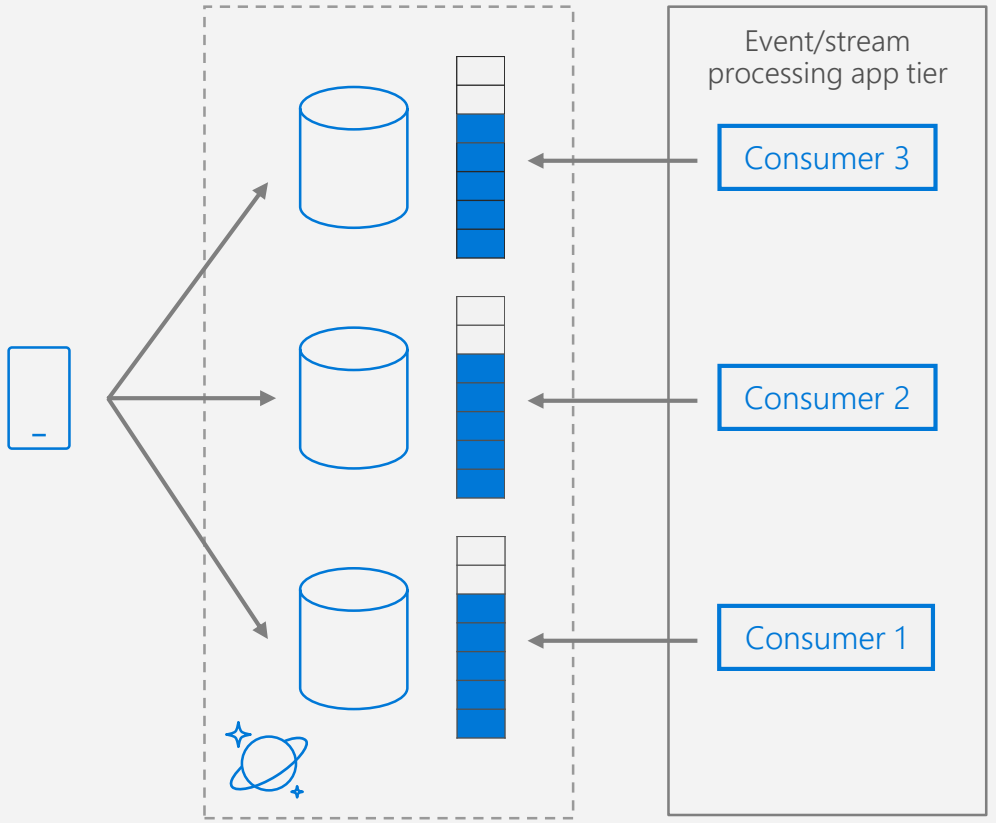


CHANGE FEED WITH PARTITIONS

Consumer parallelization

Change feed listens for any changes in Azure Cosmos DB collection. It then outputs the sorted list of documents that were changed in the order in which they were modified.

The changes are persisted, can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing. The change feed is available for each partition key range within the document collection, and thus **can be distributed across one or more consumers for parallel processing.**



CHANGE FEED PROCESSOR LIBRARY

https://www.nuget.org/packages/Microsoft.Azure.DocumentDB.ChangeFeedProcessor/

Microsoft.Azure.DocumentDB.ChangeFeedProcessor 1.3.1

Microsoft Azure Cosmos DB Change Feed Processor library

This library provides a host for distributing change feed events in partitioned collection across multiple observers. Instances of the host can scale up (by adding) or down (by removing) dynamically, and the load will be automatically distributed among active instances in about-equal way.

Package Manager | .NET CLI | Paket CLI

```
PM> Install-Package Microsoft.Azure.DocumentDB.ChangeFeedProcessor -Version 1.3.1
```

Dependencies

.NETFramework 4.5.2

Microsoft.Azure.DocumentDB (>= 1.20.2)
Newtonsoft.Json (>= 9.0.1)

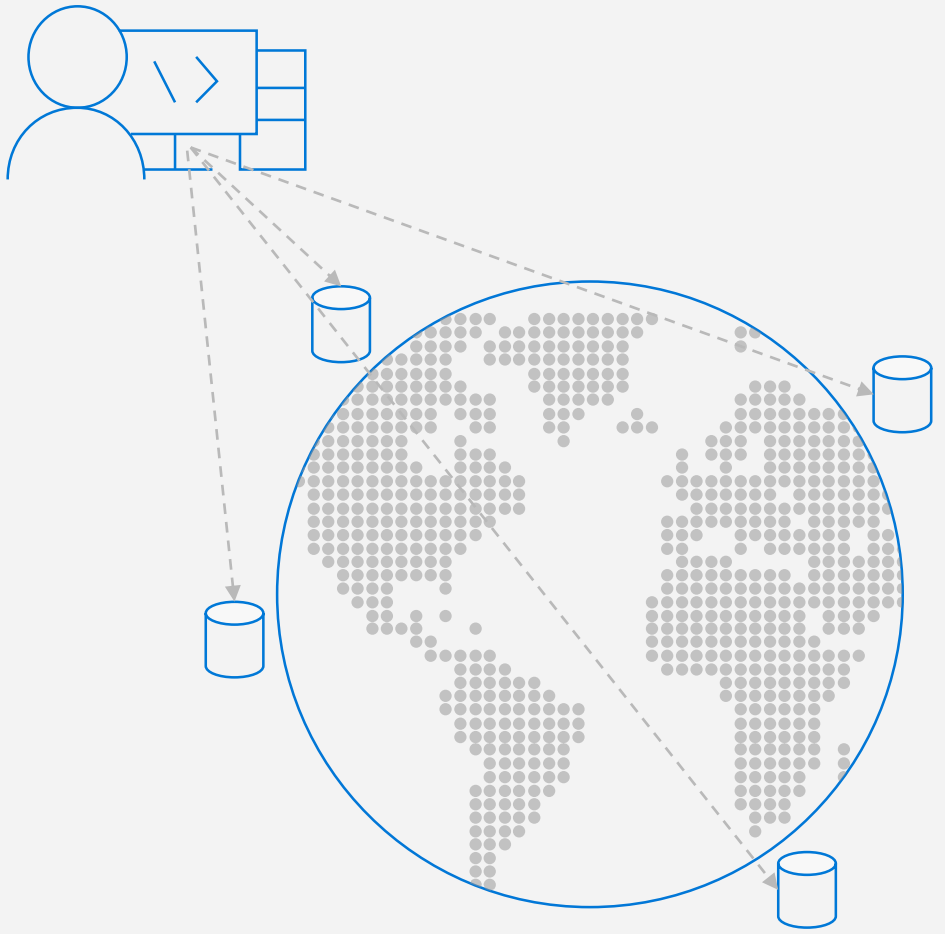
.NETStandard 2.0

Microsoft.Azure.DocumentDB.Core (>= 1.8.2)
Newtonsoft.Json (>= 9.0.1)
System.Collections.Concurrent (>= 4.3.0)

SQL SYNTAX

Using the popular query language, SQL, to access semi-structured JSON data.

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



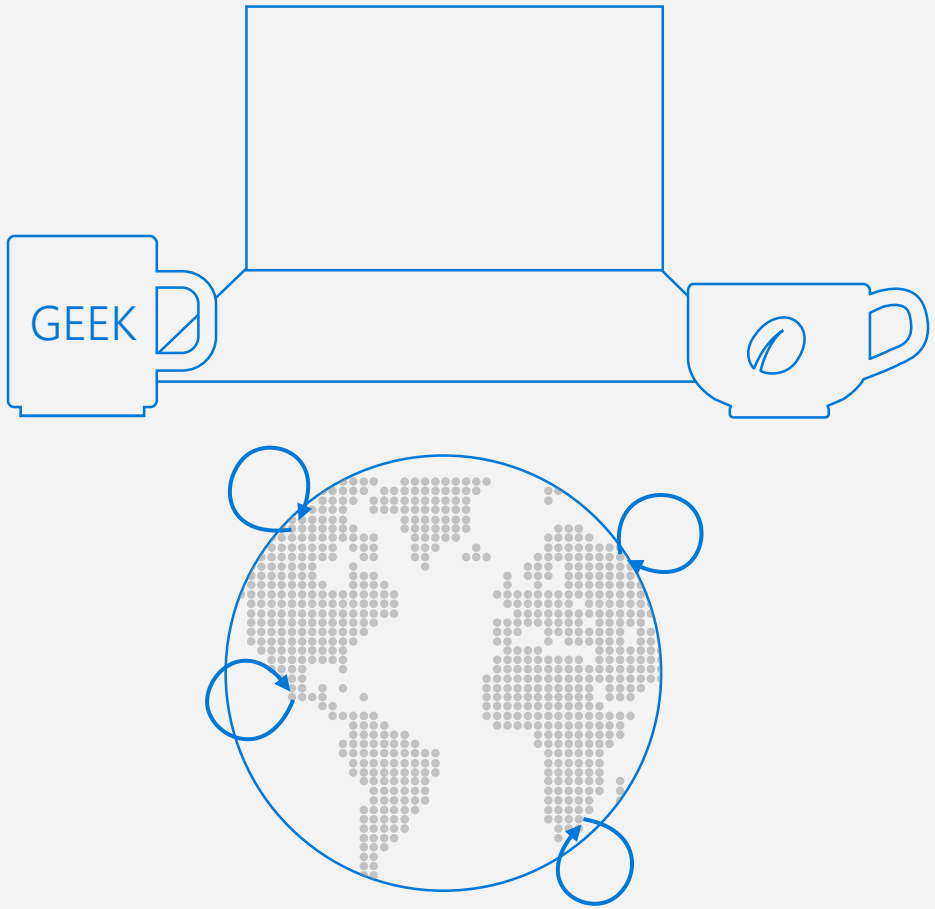
DEMO

SQL Playground

PROGRAMMING

Run native JavaScript server-side programming logic to performic atomic multi-record transactions.

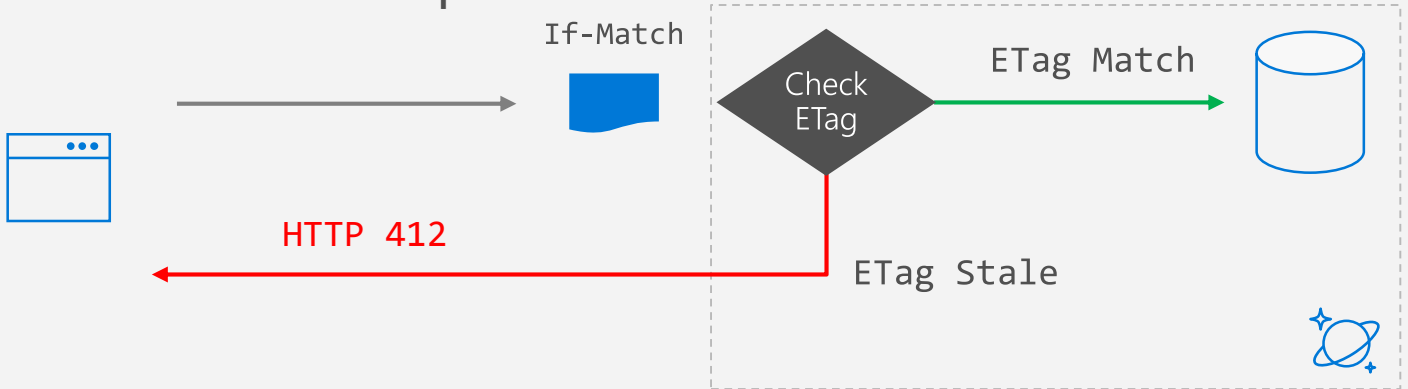
This module will reference programming in the context of the SQL API.



CONTROL CONCURRENCY USING ETAGS

OPTIMISTIC CONCURRENCY

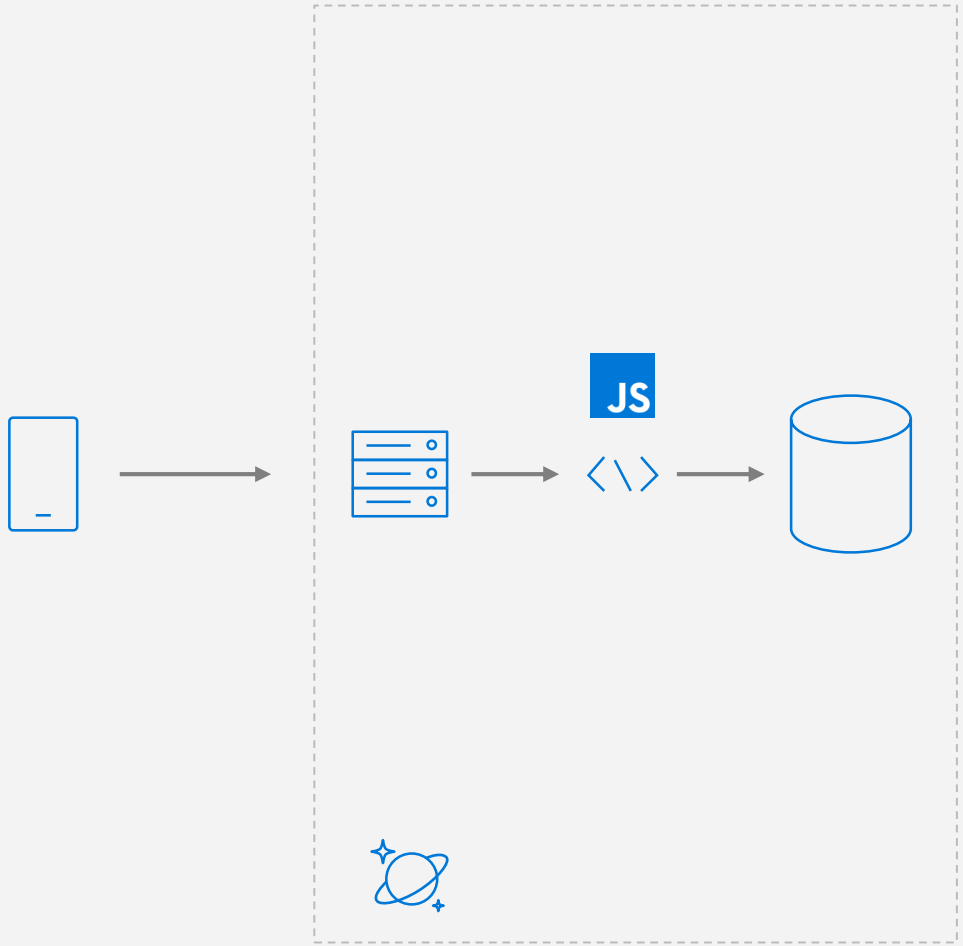
- The SQL API supports optimistic concurrency control (OCC) through HTTP entity tags, or ETags
- Every SQL API resource has an ETag system property, and the ETag value is generated on the server every time a document is updated.
- If the ETag value stays constant – that means no other process has updated the document. If the ETag value unexpectedly mutates – then another concurrent process has updated the document.
- **ETags can be used with the If-Match HTTP request header to allow the server to decide whether a resource should be updated:**



STORED PROCEDURES

BENEFITS

- Familiar programming language
- Atomic Transactions
- Built-in Optimizations
- Business Logic Encapsulation

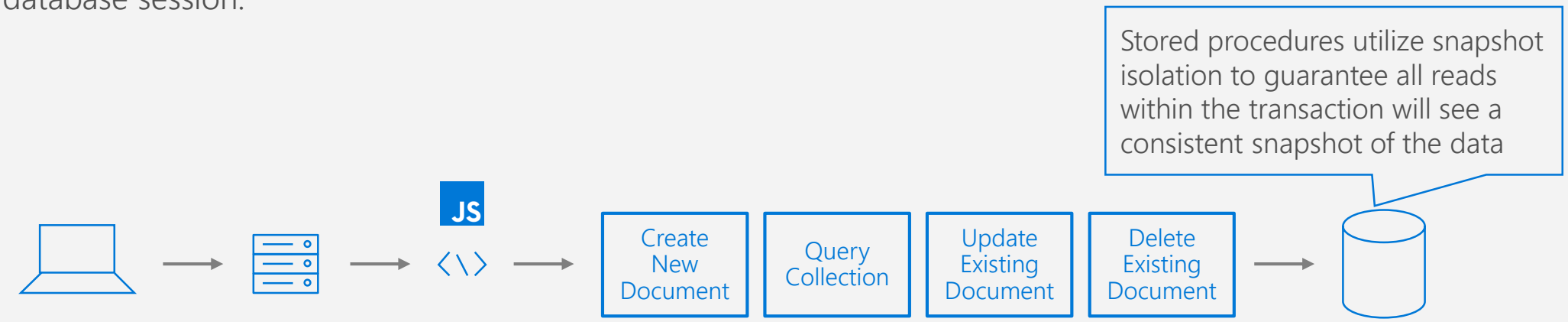


MULTI-DOCUMENT TRANSACTIONS

DATABASE TRANSACTIONS

In a typical database, a transaction can be defined as a sequence of operations performed as a single logical unit of work. Each transaction provides ACID guarantees.

In Azure Cosmos DB, JavaScript is hosted in the same memory space as the database. Hence, requests made within stored procedures and triggers execute in the same scope of a database session.



TRANSACTION CONTINUATION MODEL

CONTINUING LONG-RUNNING TRANSACTIONS

- JavaScript functions can implement a continuation-based model to batch/resume execution
- The continuation value can be any value of your own choosing. This value can then be used by your applications to **resume a transaction from a new "starting point"**