Microsoft

Azure Cosmos DB

Technical Deep Dive

Shawn Weisfeld
sweisfel@microsoft.com
April 29 2019

Modern Data Platform Requirements

Global distribution

Scalability for PB-sized data, trillions of ops/sec

Guaranteed low latency at any scale

Fine-grained Elasticity

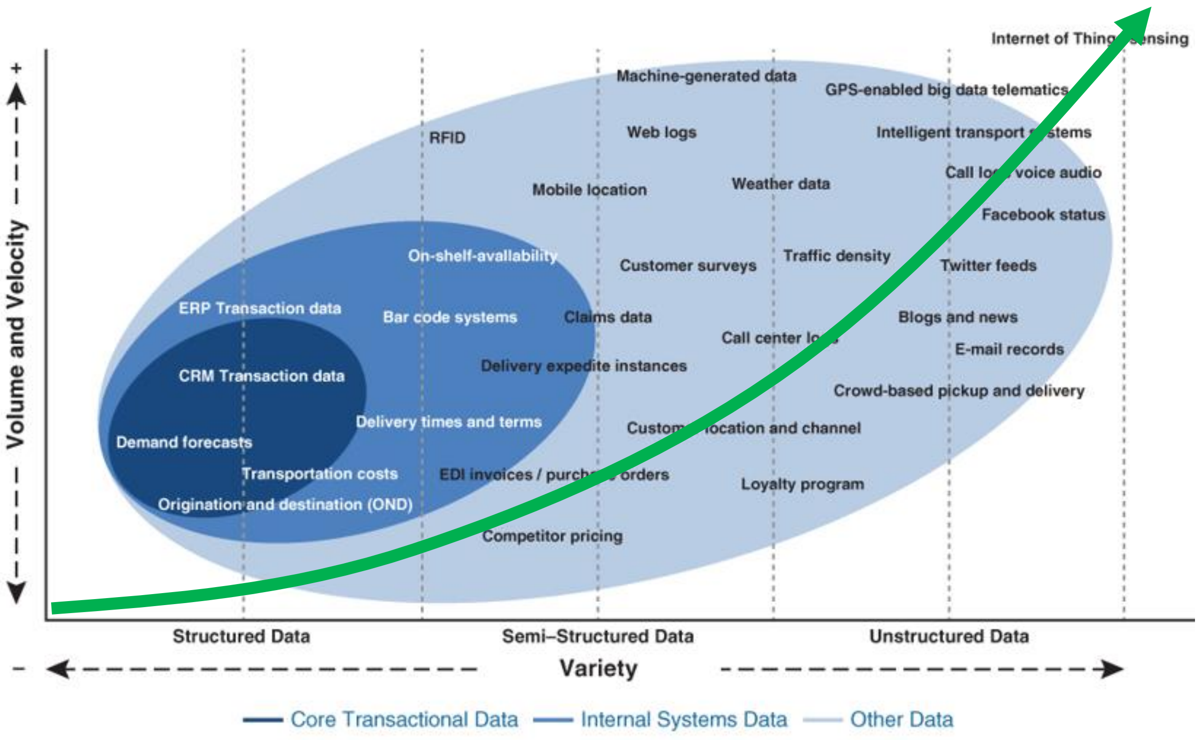
High availability

Seamless support for data heterogeneity & schema evolution

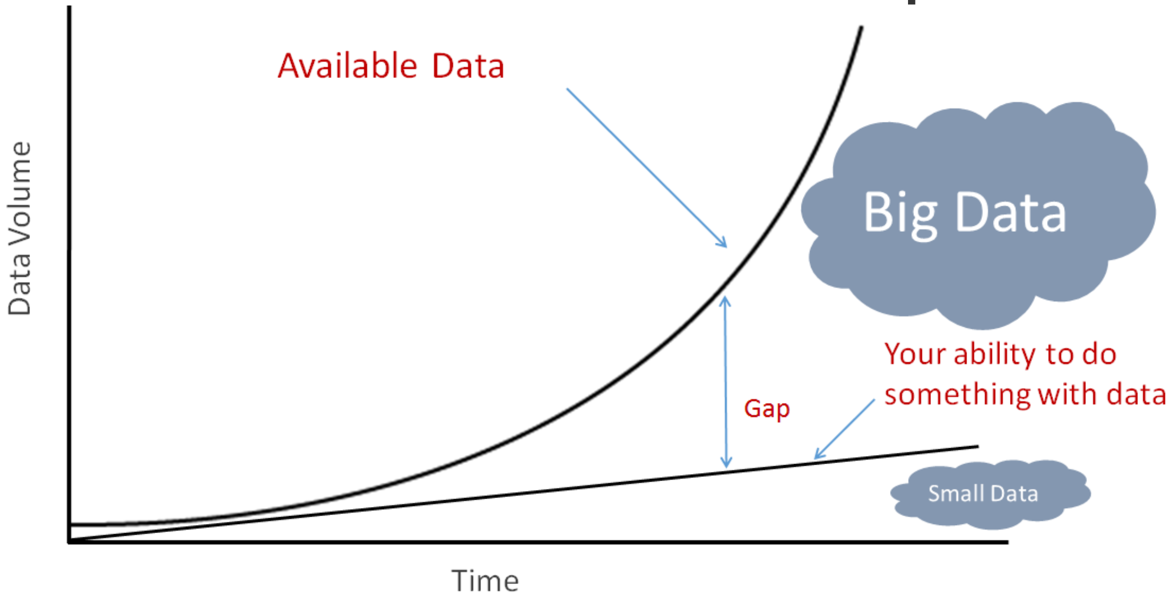
Fast time-to-insight

We need a completely different way of managing data!

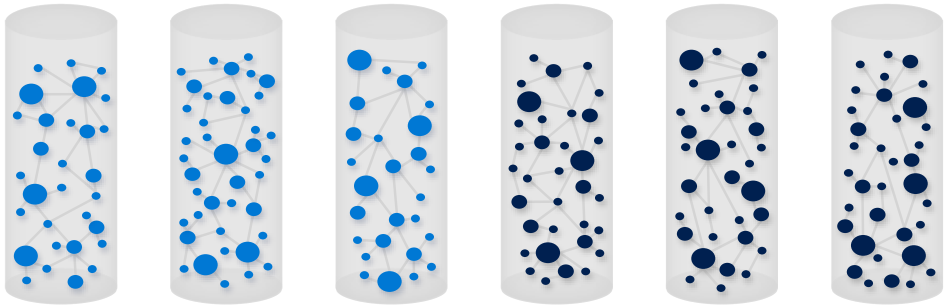
Data is the Competitive Advantage



Data Explosion

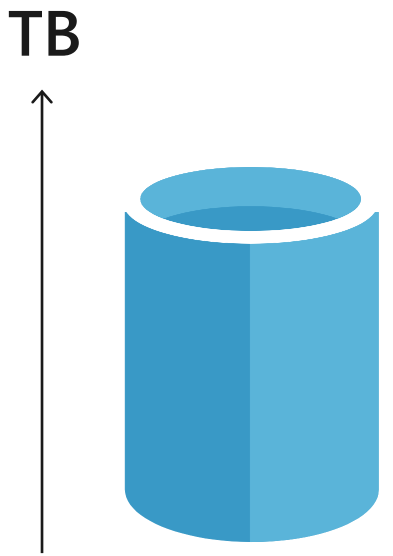


Internal and data external silos



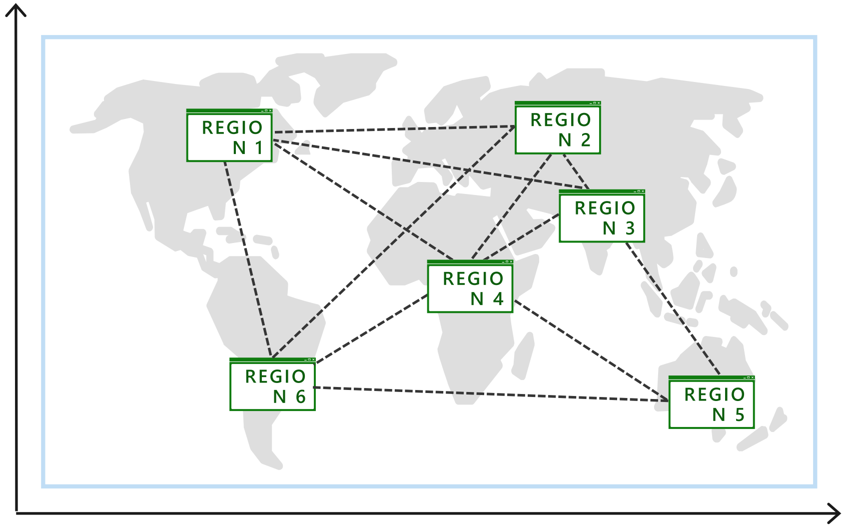
Mobile/Web Transactions IoT Bottlers Regions BUs

3 Technology Choices



Scale up, relational

Scale out, Blob storage



Scale out, globally distributed

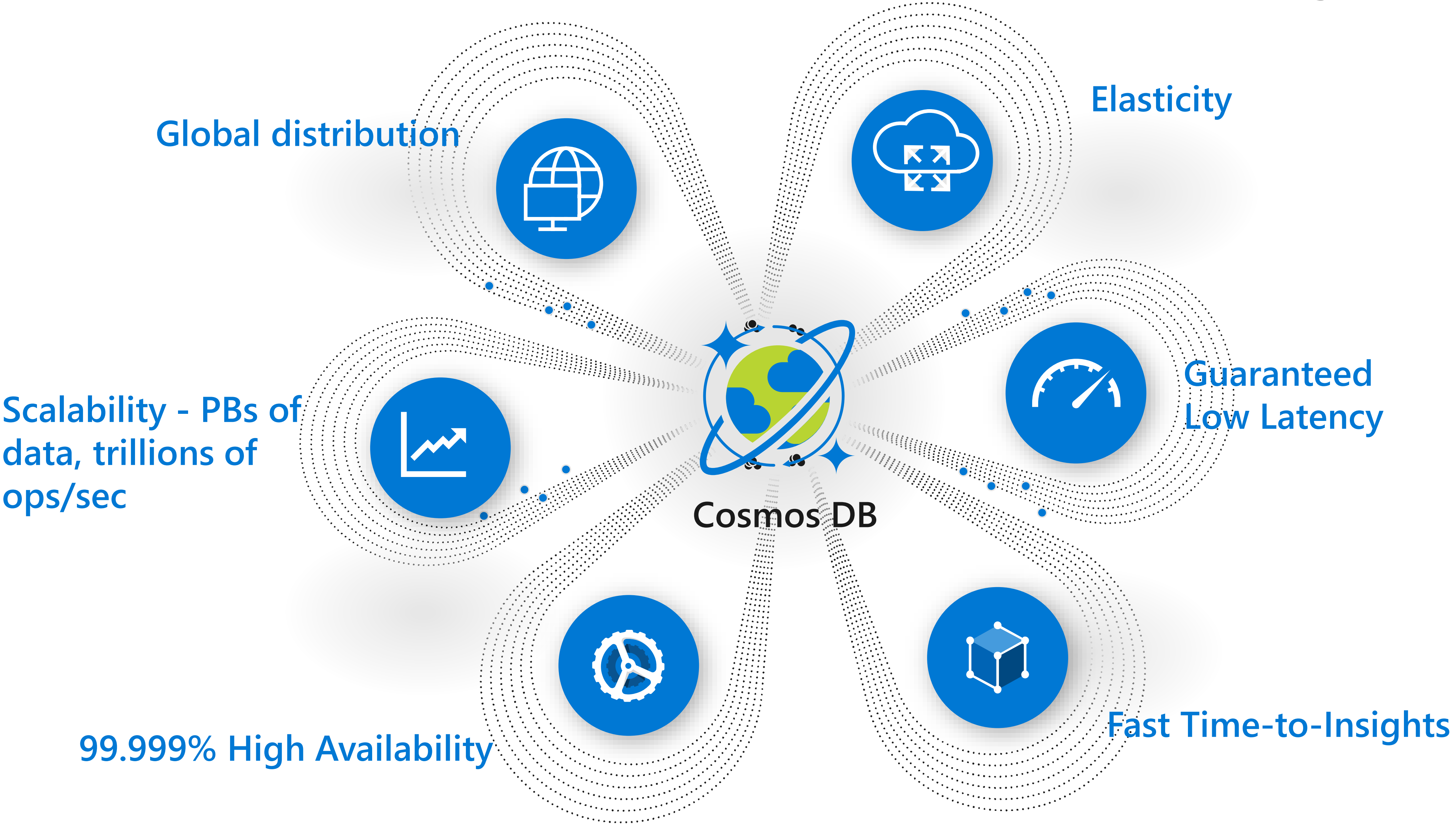
Low TBs of data
Thousands of operations/sec
Rich indexing and querying capabilities
Requires schemas

Petabytes of data
Thousands of operations/sec
Schema-agnostic

Low latency ✓
Petabytes of data ✓
Trillions of operations/sec ✓
Global distribution ✓
Rich indexing and querying capabilities ✓
Schema-agnostic ✓

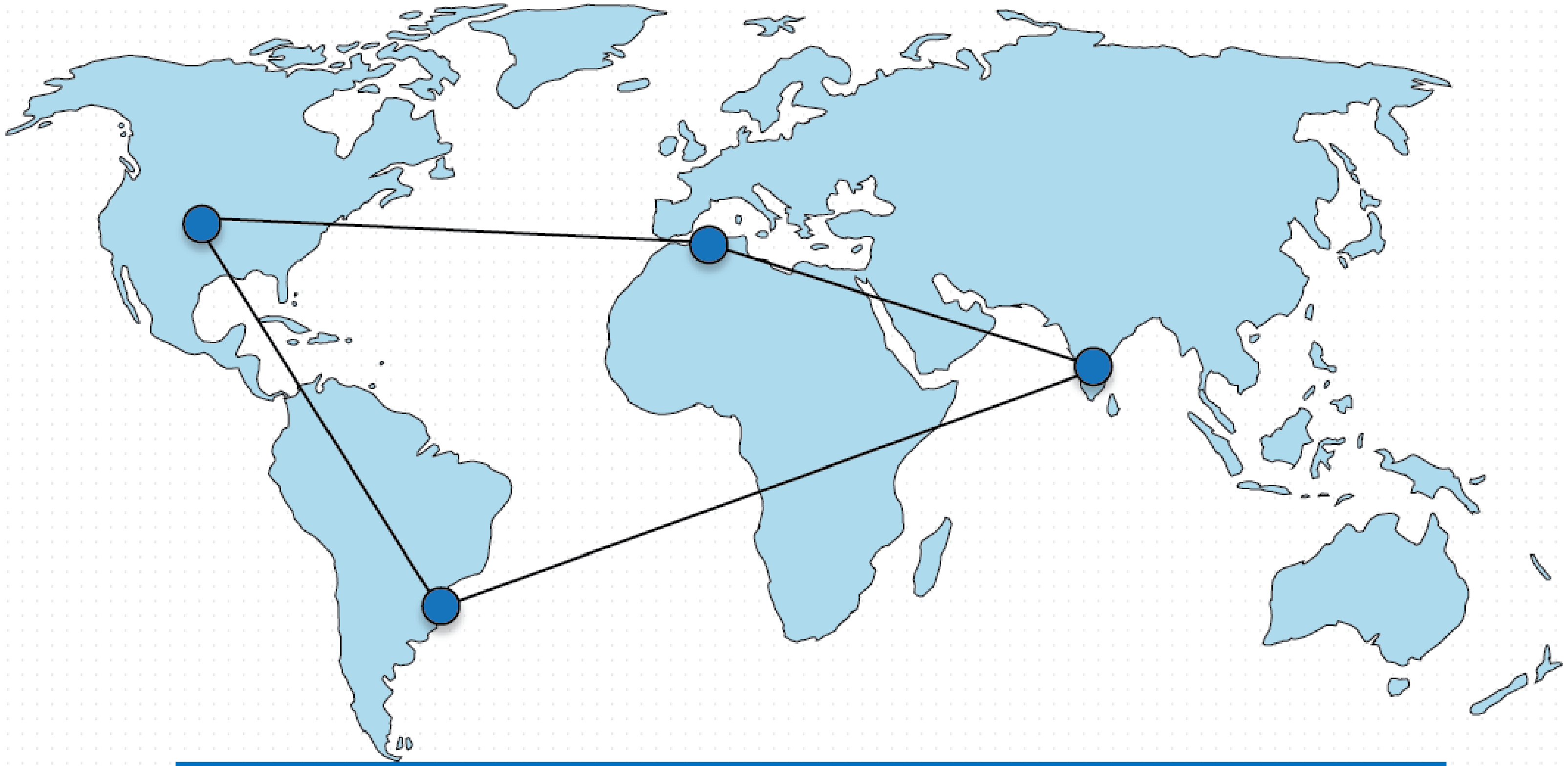
Cosmos DB

Customers need a database that exploits the cloud's design center



Azure Cosmos DB

Table | SQL | JavaScript | API for MongoDB | Gremlin | Cassandra | Spark | ETCD | ...more APIs coming



Key-Value | Column-Family | Documents | Graph

Global distribution | Elastic scale-out | Guaranteed low latency | Five consistency models | Comprehensive SLAs

Most Secure and Compliant

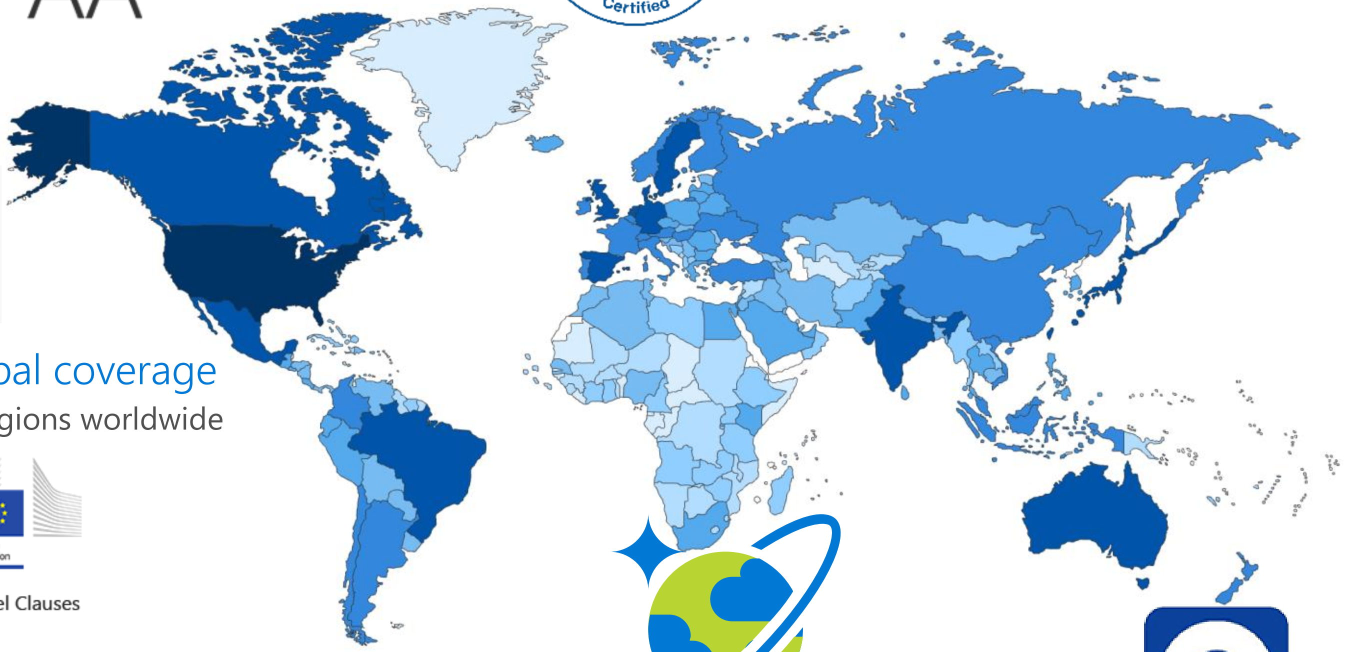
HIPAA

ISO 27001 Certified

CSA cloud security alliance®

GOV.UK

PCI Security Standards Council™



Global coverage

54 regions worldwide

European Commission

EU Model Clauses

Federal Office for Information Security

National Clouds

US Government

China

Germany

FedRAMP

Cosmos DB

China GB 18030

Geo-fencing

Custom role-based access control

Encryption-at-rest
Encryption-in-motion

IP Filtering

VNET

Cosmos DB is named a leader in the Forrester Wave™ 2019: Big Data NoSQL

We're excited to announce that Forrester has named Microsoft as a **Leader** in The Forrester Wave™: Big Data NoSQL, Q1 2019 based on their evaluation of Azure Cosmos DB. We believe Forrester's findings validate the exceptional market momentum of Azure Cosmos DB and how happy our customers are with the product.

NoSQL platforms are on the rise

According to Forrester, "half of global data and analytics technology decision makers have either implemented or are implementing NoSQL platforms, taking advantage of the benefits of a flexible database that serves a broad range of use cases…While many organizations are complementing their relational databases with NoSQL, some have started to replace them to support improved performance, scale, and lower their database costs."

Azure Cosmos DB has market momentum

Azure Cosmos DB is Microsoft's globally distributed, multi-model database service for mission-critical workloads. Azure Cosmos DB provides turnkey global distribution with unlimited endpoint scalability, elastic scaling of throughput (at multiple granularities, e.g., database, key-space, tables and collections) and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency models, and guaranteed high availability, all backed by the industry-leading comprehensive SLAs. Azure Cosmos DB automatically indexes all data without requiring developers to deal with schema or index management. It is a multi-model service, which natively supports document, key-value, graph, and column-family data models. As a natively born in the cloud service, Azure Cosmos DB is carefully engineered with multitenancy and global distribution from the ground up. As a foundational service in Azure, Azure Cosmos DB is ubiquitous, running in all public regions, DoD and sovereign clouds, with industry-leading compliance certification list, enterprise grade security – all without any extra cost.

According to the Forrester report, Azure Cosmos DB is starting to achieve a really strong traction and ***"Its simplified database with relaxed consistency levels and low-latency access makes it easier to develop globally distributed apps." Forrester mentioned specifically that "Customer references like its resilience, low maintenance, cost effectiveness, high scalability, multi-model support, and faster time-to-value."***

Industry-Leading Enterprise Customers using Cosmos DB

jet · Walmart · asos discover fashion online · Domino's Pizza · Johnson & Johnson · Coca-Cola · MARC JACOBS · Emirates

ABB · Deloitte. · Johnson Controls · Ford · CRESTRON · SIEMENS · sitecore Own the experience · Sam's CLUB · Schneider Electric

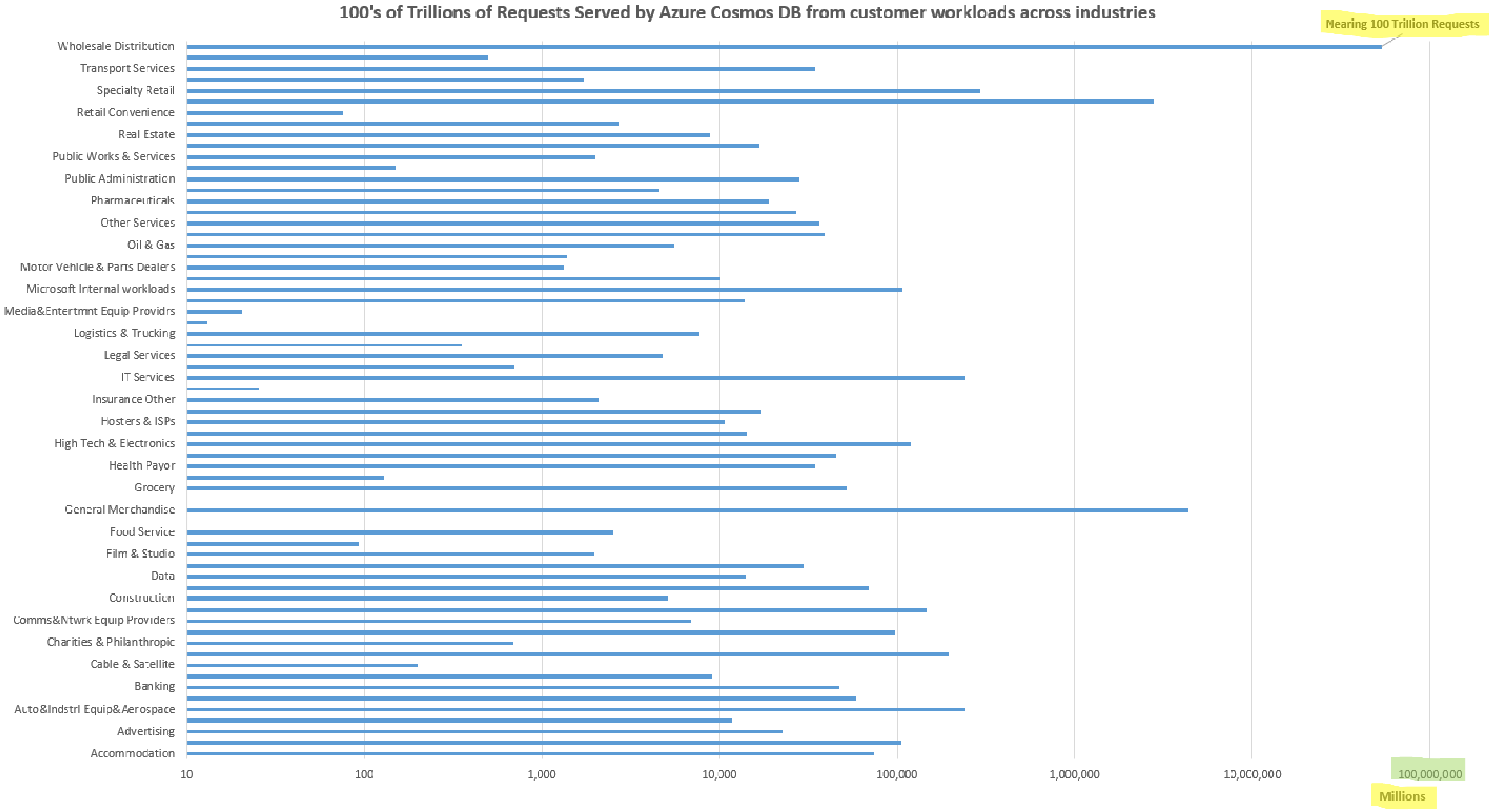
TOYOTA · Rakuten · Adobe · GEP · WARNER BROS. PICTURES · Rockwell Automation · INMOBI · Bb Blackboard · ERICSSON

CITRIX · Diply · Unilever · GE · esri · Alaska · BEST BUY · Bentley

BOEING · Honeywell · Rolls-Royce · Wolters Kluwer · CATERPILLAR · pwc · DnB · TESCO · VOLVO

Microsoft · LinkedIn · XBOX · Office 365 · Skype · Azure · GitHub

100’s of Trillions of Requests served by Azure Cosmos DB with high 99.999+ SLA, workloads across all industries. Azure Cosmos DB most favored by retailers with wholesalers themselves nearing 100 trillion requests



Jet.com on Cosmos DB | Panther

"Data at Jet.com and Walmart Labs is scale. E-commerce will be a $1 trillion business in the next five to ten years, and Azure Cosmos DB is critical to our vision, because we need a database that will elastically scale with us. Not only that – but we also need true elasticity and the ability to scale up/down on demand to help us manage our costs carefully. We experience big peaks during key shopping days, where expected rates of events can increase 10 to 20 compared to the rest of the year.

For example, for Black Friday and Cyber Monday – we needed the ability to enough provisioned throughput to satisfy 1 trillion Request Units over 24 hours to satisfy our customer demand. To solve this – we deployed a geo-replicated Cosmos DB collection configured for 10 million request units per second.

Another key aspect for us is performance. We have rigorous latency objectives to ensure a responsive customer experience – in which we really like Cosmos DB's predictable latency characteristics where we observe single-digit millisecond latency reads and writes.

Cosmos DB's elastic scale, automatic indexing and Change Feed support makes it an excellent event store for event sourcing. Heterogeneous events can be automatically indexed for query, and the event log exposed through Change Feed can be used to reconstruct past states. Performing writes in an append-only manner via event sourcing is fast and scalable for systems that are write-heavy, in particular when used in combination with a write-optimized database. Many of our microservices are mission critical components to our business. Cosmos DB gives us peace of mind with an aggressive 99.99% availability guarantee when deployed to a single region – which is further strengthened to 99.999% when deployed across regions using Cosmos DB's global distribution."

- Scott Havens, Director of Engineering, Jet.com and Walmart Labs

Walmart HVAC IoT on Cosmos DB

Challenge

Walmart has over 11,ooo stores globally, with HVAC sensors in each store capturing telemetry and triggering repair actions when aberrations are encountered.

Strategy

Cosmos DB is being used to ingest large volumes of sensor readings, with each store periodically emitting HVAC readings in 5 minute intervals.

Results

- The throughput and latency have been very consistent, with Cosmos DB capturing nearly 3B sensor readings per day.
- Total Throughput – 1.3M operations/second
- Regions – Currently 2 (West US, East US 2), more regions upcoming

"Cosmos DB has delivered a high-performance worldwide platform for our global reputation system. With a click of a button we can rapidly deploy regions worldwide to deliver excellent response time to our customers and partners. The Cosmos DB Cassandra API eliminated major engineering efforts during migration of our services as we were able to integrate with very minimum change. Our largest service is slated to migrate with the launch of Cassandra API support."

Michael Shavell, Technical Director / Architect at Symantec

"Azure Cosmos DB is an amazing technology. The biggest benefit for us is that we can have one database for anonymous data that is replicated worldwide into all regions that are relevant for us, in a consistency model that perfectly fits our needs. It's key that Azure Cosmos DB is naturally built into the Azure core infrastructure, and we can build on its high availability and replication, offering software to all our customers at the same time."

Thomas Gossler: Lead Architect, Digital Ecosystem Platform, Siemens Healthineers

"At Liberty Mutual, we are always looking for new ways to engage with our customers and personalize their web experience. To do this, we needed to approach data modeling differently. We had a mix of hierarchical and transitive relationship patterns and data normalization was only taking us so far. Modeling our data as a graph enabled us to address our needs, and Azure Cosmos DB enabled us to deliver this solution quicker, and in a cloud managed environment that's enterprise-ready, scalable and highly performant. The Cosmos DB Change Feed API helped us simplify synchronization with existing Systems of Record, allowing us to focus on building customer-centric capabilities rather than infrastructure and operational concerns."

Arvin Suresh, Architect, Liberty Mutual Insurance

discover fashion online

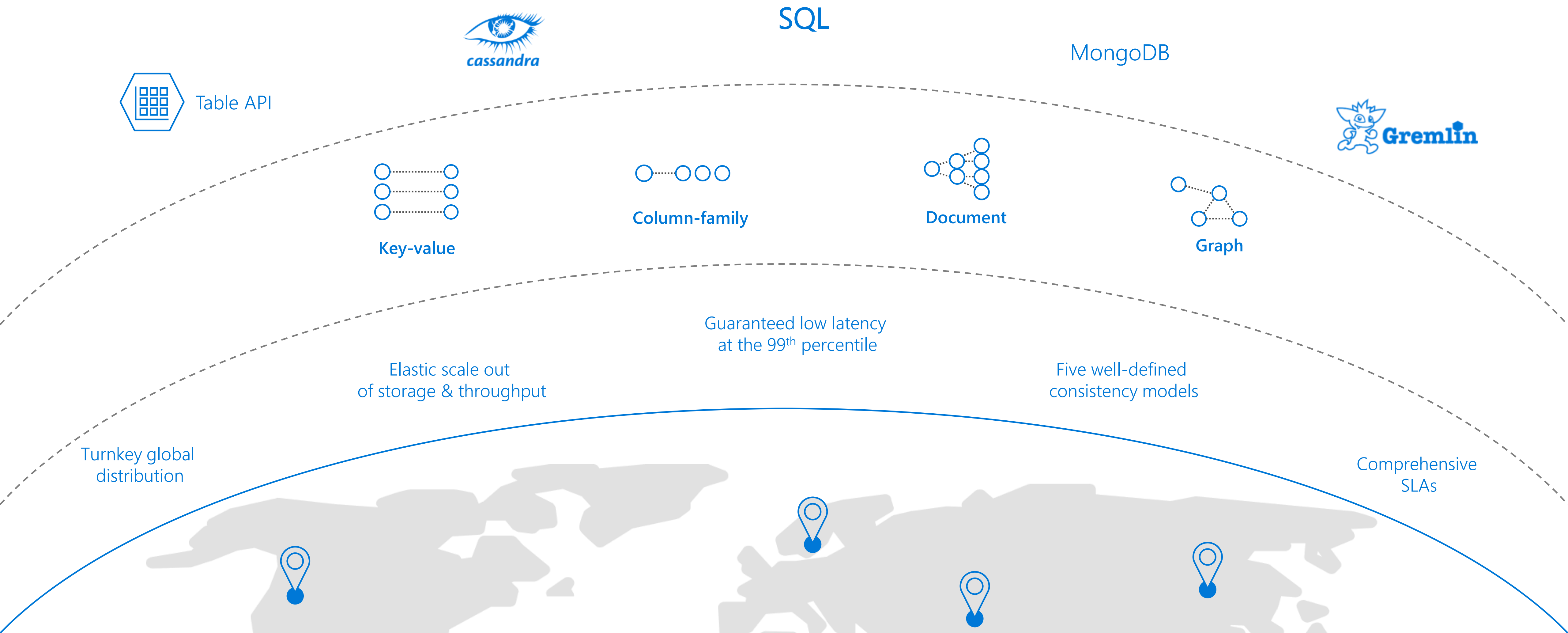
"At ASOS, we experience tremendous customer demand handling >100M unique daily visitors across >200 countries shopping over >100K of products. We needed a distributed database that seamlessly and elastically scales to serve our global customers. During peak sales periods we need extra computing power to cope with customer demand and manage our large datasets automatically behind the scenes without the burden of complex online data migrations – freeing our engineering teams to concentrate on business features, reducing the need for complex coordination with business + operations teams, and most importantly, seamlessly scaling with zero planned downtime. Well, I have to say I've been impressed with Azure Cosmos DB - we've watched our collections seamlessly grow as our applications have grown while maintaining its 99.999% availability SLA."

- Gary Strange, Lead Data Engineer, ASOS

Cosmos DB

AZURE COSMOS DB

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

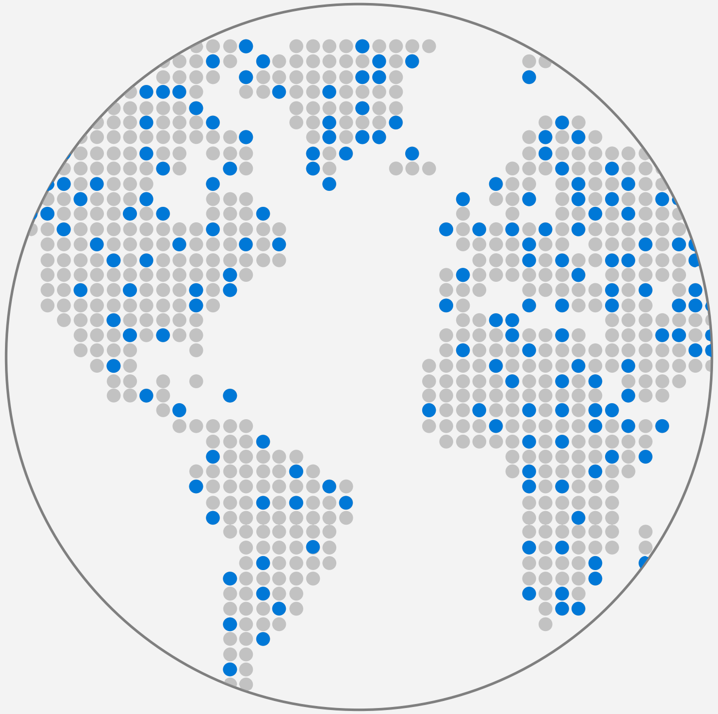


TURNKEY GLOBAL DISTRIBUTION

PUT YOUR DATA WHERE YOUR USERS ARE

Automatically replicate all your data around the world, and across more regions than Amazon and Google combined.

- Available in all Azure regions
- Manual and automatic failover
- Automatic & synchronous multi-region replication

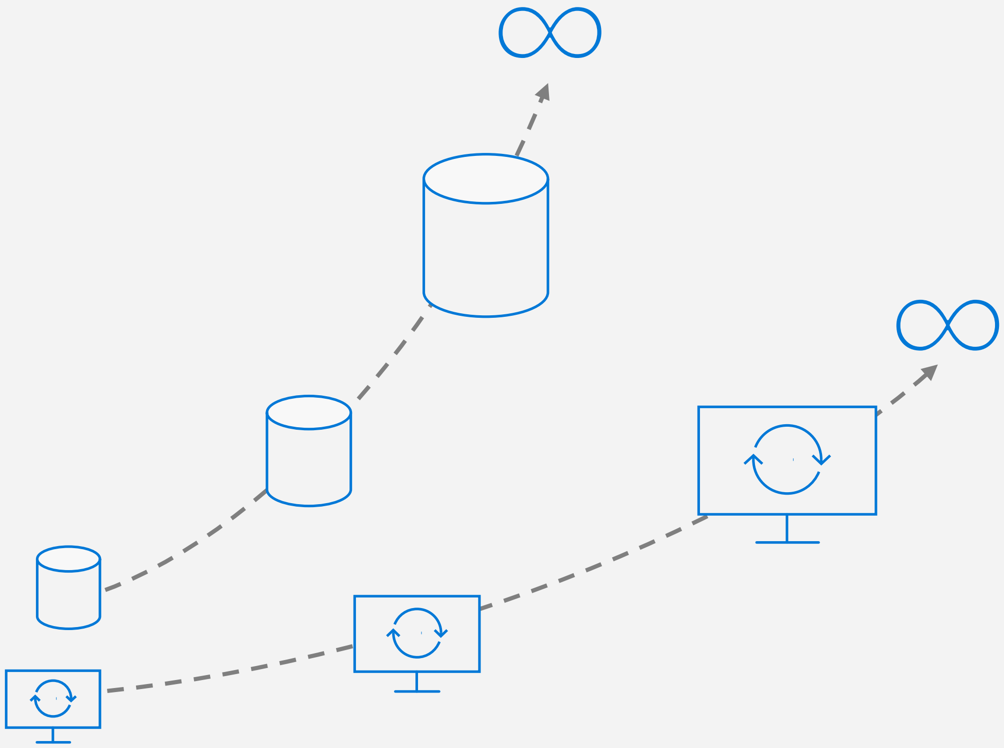


ELASTIC SCALE OUT OF STORAGE AND THROUGHPUT

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.

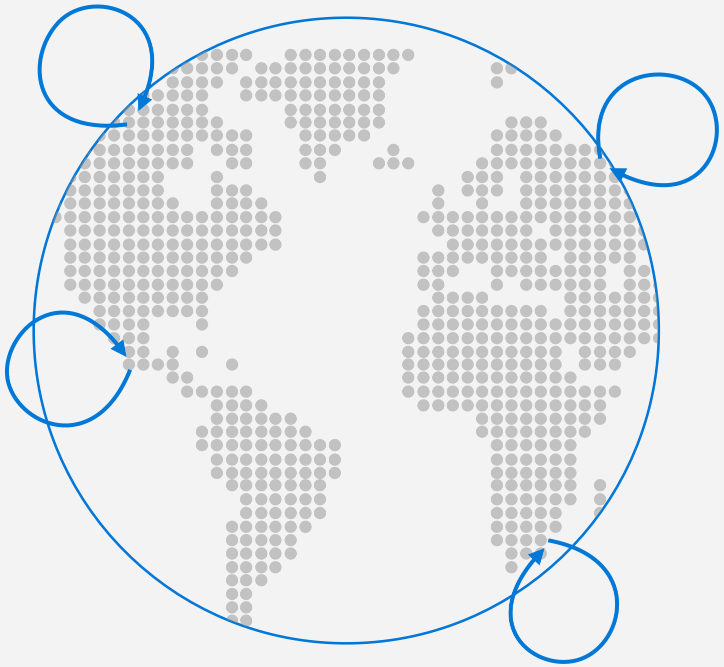
- Elastically scale throughput from 10 to 100s of millions of requests/sec across multiple regions
- Support for requests/sec for different workloads
- Pay only for the throughput and storage you need



GUARANTEED LOW LATENCY

PROVIDE USERS AROUND THE WORLD WITH FAST ACCESS TO DATA

Serve <10 ms read and <15 ms write requests at the 99th percentile from the region nearest to users, while delivering data globally.



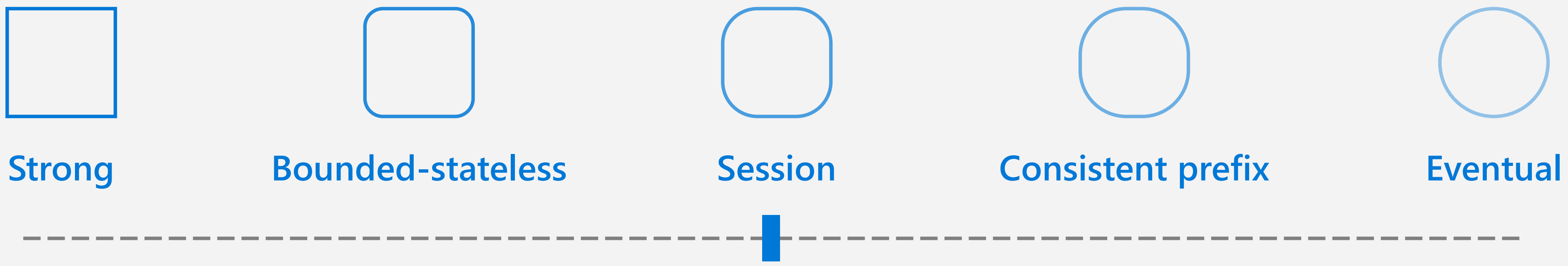
FIVE WELL-DEFINED CONSISTENCY MODELS

CHOOSE THE BEST CONSISTENCY MODEL FOR YOUR APP

Offers five consistency models

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.



MULTIPLE DATA MODELS AND APIs

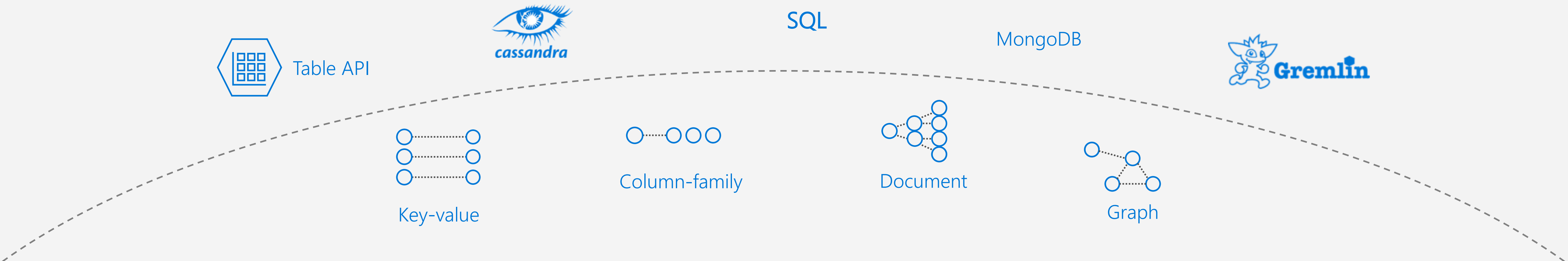
USE THE MODEL THAT FITS YOUR REQUIREMENTS, AND THE APIS, TOOLS, AND FRAMEWORKS YOU PREFER

Cosmos DB offers a multitude of APIs to access and query data including, SQL, various popular OSS APIs, and native support for NoSQL workloads.

Use key-value, tabular, graph, and document data

Data is automatically indexed, with no schema or secondary indexes required

Blazing fast queries with no lag

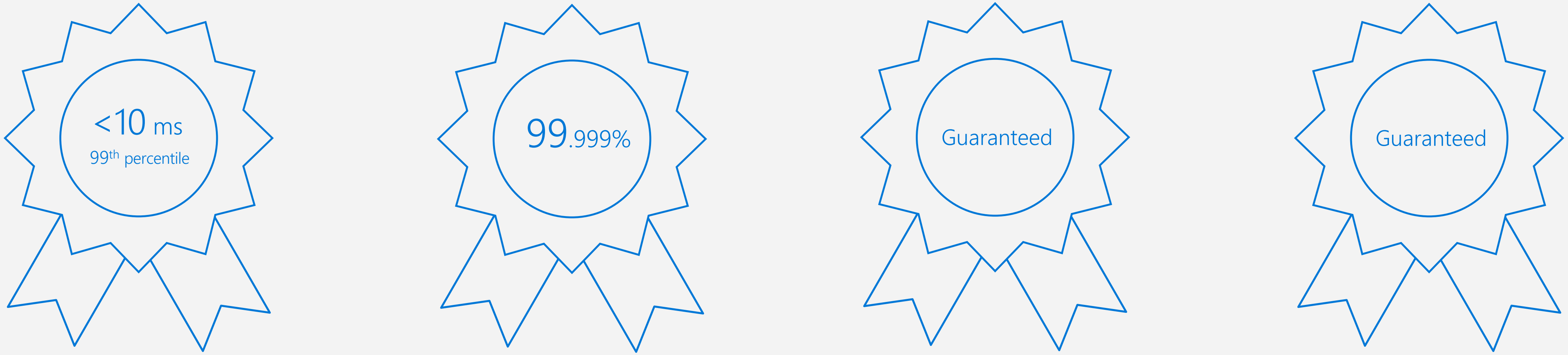


COMPREHENSIVE SLAs

RUN YOUR APP ON WORLD-CLASS INFRASTRUCTURE

Azure Cosmos DB is the only service with financially-backed SLAs for millisecond latency at the 99th percentile, 99.999% HA and guaranteed throughput and consistency

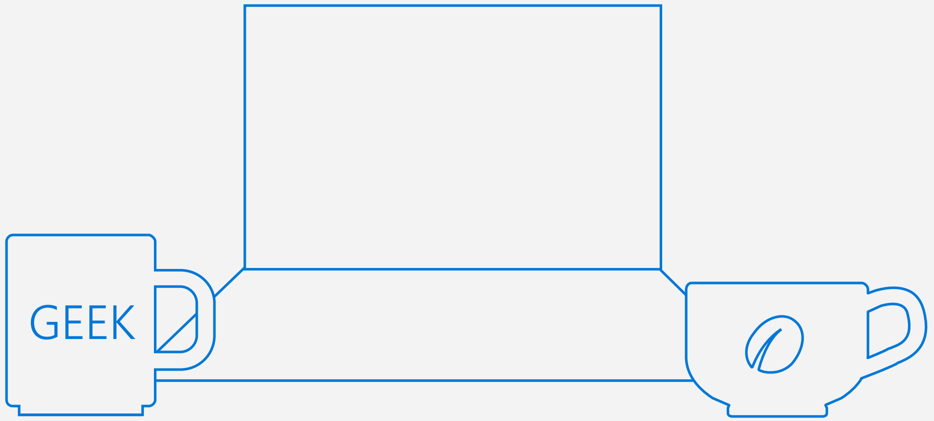
Latency | **HA** | **Throughput** | **Consistency**



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

TRUST YOUR DATA TO INDUSTRY-LEADING SECURITY & COMPLIANCE

Azure is the world's most trusted cloud, with more certifications than any other cloud provider.

- Enterprise grade security
- Encryption at Rest
- Encryption is enabled automatically by default
- Comprehensive Azure compliance certification

DEMO

ToDo App

https://github.com/shawnweisfeld/CosmosToDo

TODO DEMO

- Show Web App
- Show Global Data Replication – Add region.
- Show Consistency
- Show Data Explorer
  - Scale
  - SPs, USFs, Triggers
  - Document Query

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

TOP 10 REASONS WHY CUSTOMERS USE AZURE COSMOS DB

The 1^{st} and only database with global distribution turnkey capability

Deliver massive storage/throughput scalability database

Provides guaranteed single digit millisecond latency at 99^{th} percentile worldwide

Natively supports different types of data at massive scale

Boasts 5 well-defined consistency models to pick the right consistency/latency/ throughput tradeoff

Enables mission critical intelligent applications

Gives high flexibility to optimize for speed and cost

Tackles big data workloads with high availability and reliability

Provides multi-tenancy and enterprise-grade security

Naturally analytics-ready and perfect for event-driven architectures

AZURE COSMOS DB

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

- Multi-model data with your favorite API
- Elastically scale storage and throughput
- Multiple, well-defined consistency levels
- <10ms latency guarantees at the 99th percentile
- Industry-leading SLAs across performance, latency, availability and throughput

Lift and shift MongoDB apps

Run Spark over operational data

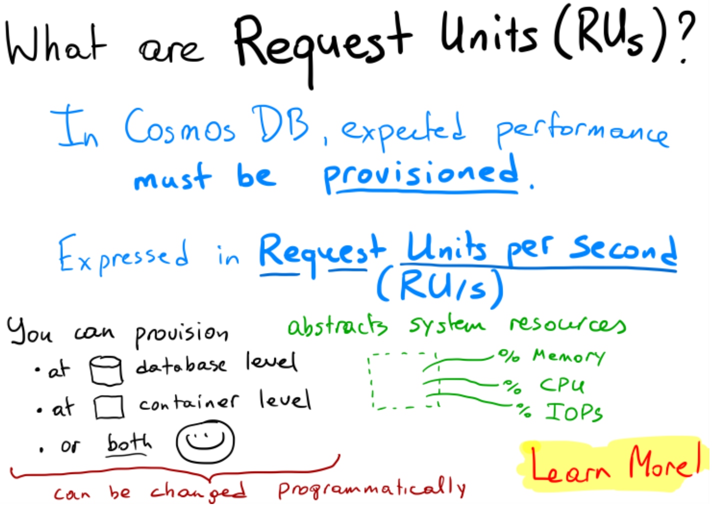
Build real-time customer experiences

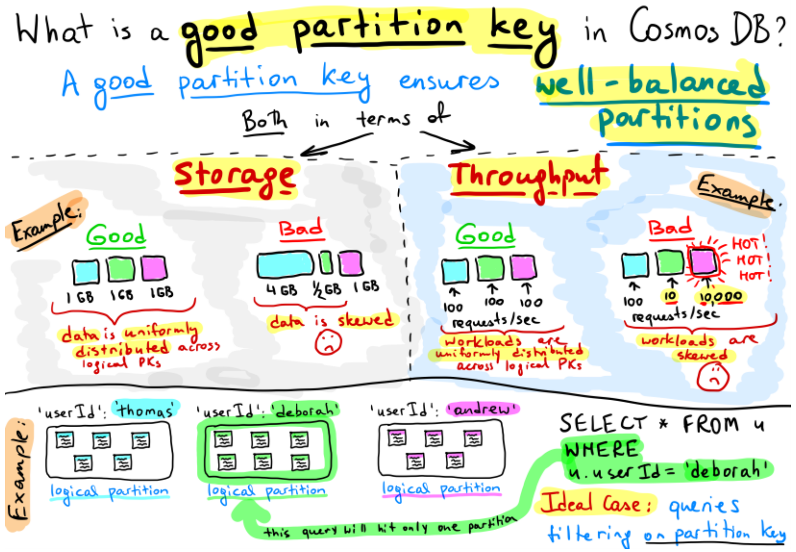
Ideal for IoT, gaming and eCommerce

More Info

- Getting Started - https://www.gotcosmos.com/

- Cosmic Notes - https://azurecosmosdb.github.io/CosmicNotes/





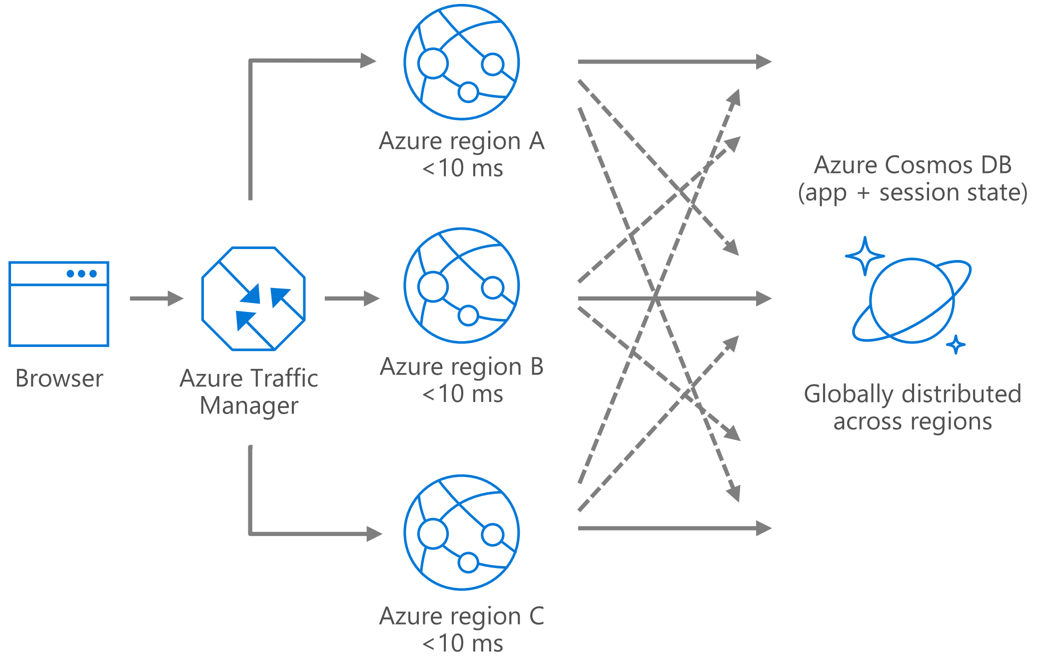
Microsoft

App Patterns and Use Cases

DATA DISTRIBUTED AND AVAILABLE GLOBALLY

Put your data where your users are to give real-time access and uninterrupted service to customers anywhere in the world.

- Turnkey global data replication across all Azure regions
- Guaranteed low-latency experience for global users
- Resiliency for high availability and disaster recovery



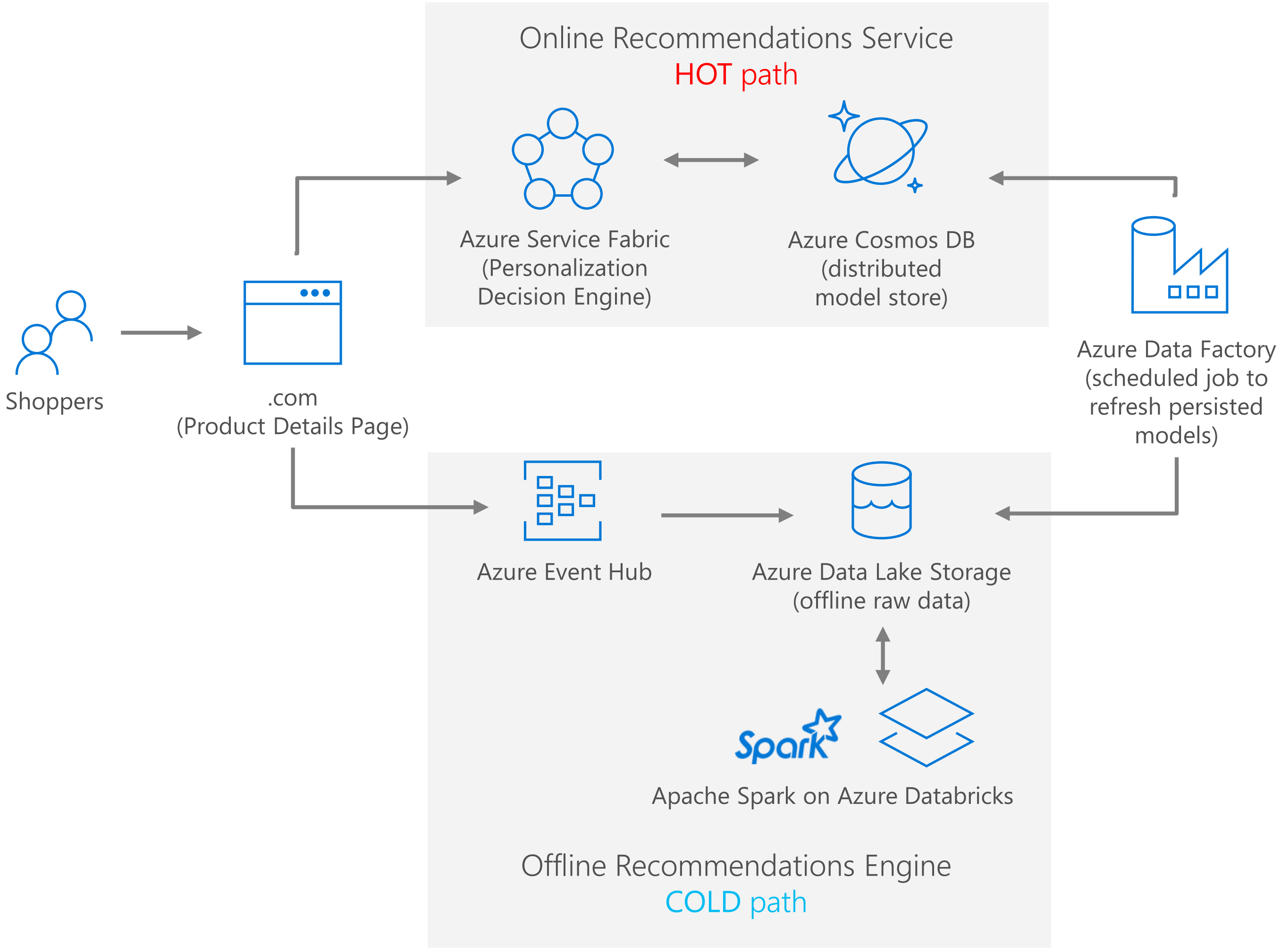
NEXT GAMES

Domino's

BUILD REAL-TIME CUSTOMER EXPERIENCES

Offer latency-sensitive applications with personalization, bidding, and fraud-detection.

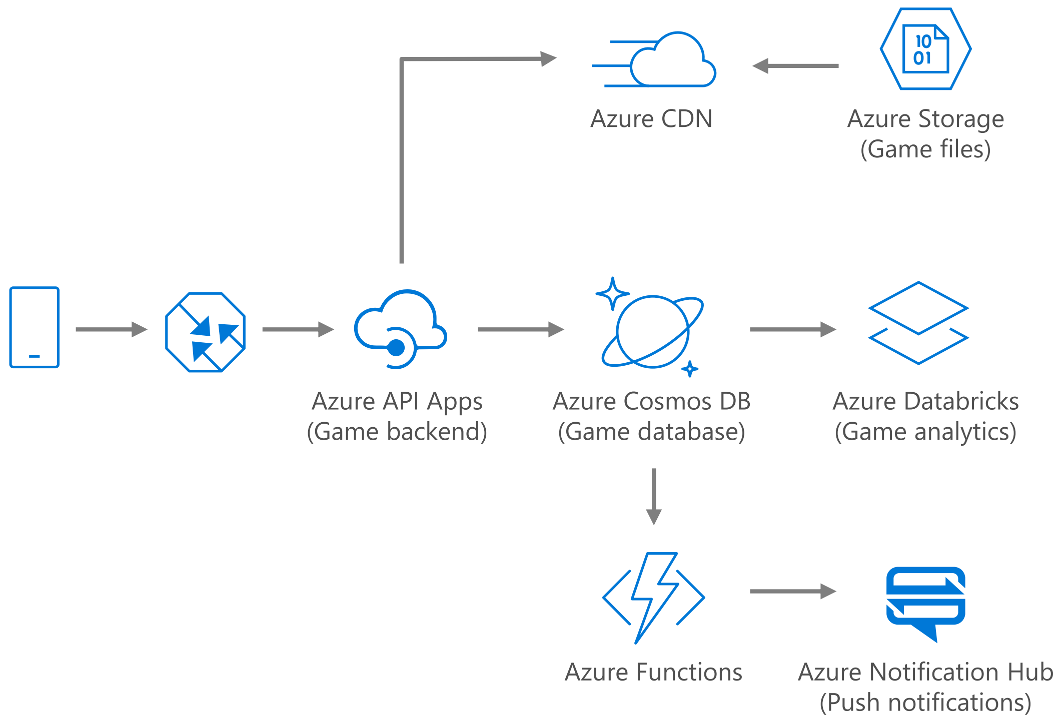
- Machine learning models generate real-time recommendations across product catalogues
- Product analysis in milliseconds
- Low-latency ensures high app performance worldwide
- Tunable consistency models for rapid insight



IDEAL FOR GAMING, IOT AND ECOMMERCE

Maintain service quality during high-traffic periods requiring massive scale and performance.

- Instant, elastic scaling handles traffic bursts
- Uninterrupted global user experience
- Low-latency data access and processing for large and changing user bases
- High availability across multiple data centers

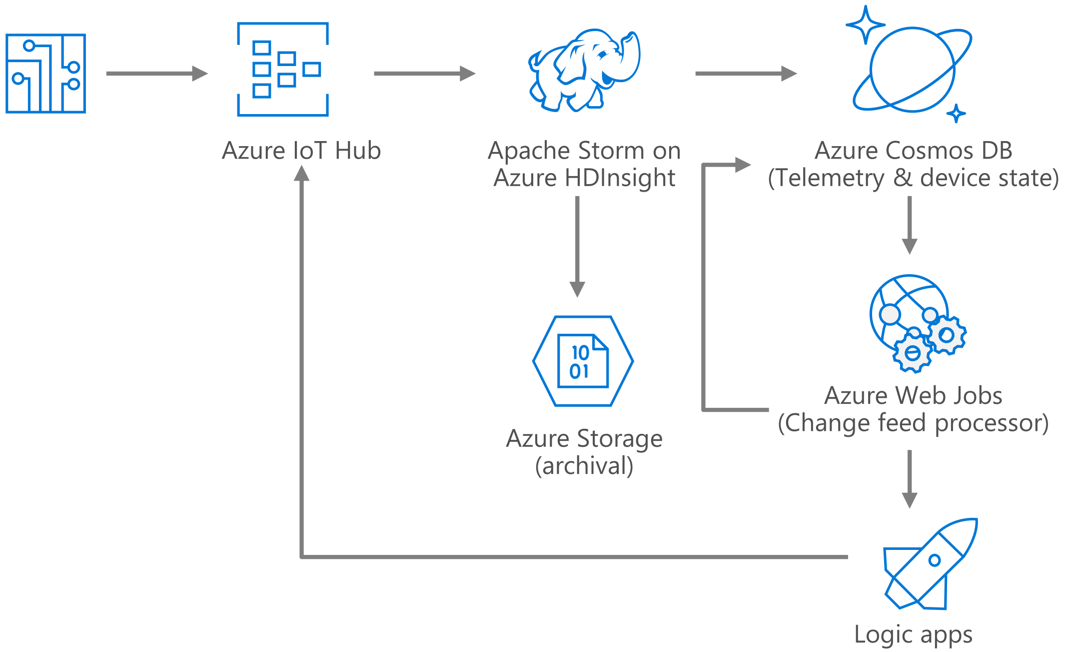


NEXT GAMES Domino's Jet XBOX LIVE

MASSIVE SCALE TELEMETRY STORES FOR IOT

Diverse and unpredictable IoT sensor workloads require a responsive data platform

- Seamless handling of any data output or volume
- Data made available immediately, and indexed automatically
- High writes per second, with stable ingestion and query performance

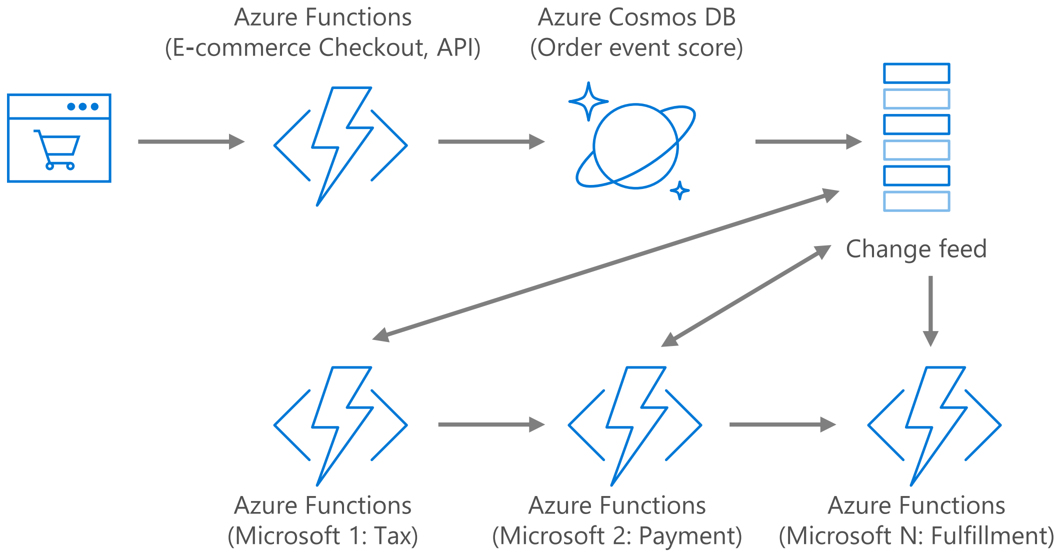


TOYOTA Honeywell LG CNS Johnson Controls

SIMPLIFIED DEVELOPMENT WITH SERVERLESS ARCHITECTURE

Experience decreased time-to-market, enhanced scalability, and freedom from framework management with event-driven micro-services.

- Seamless handling of any data output or volume
- Data made available immediately, and indexed automatically
- High writes per second, with stable ingestion and query performance
- Real-time, resilient change feeds logged forever and always accessible
- Native integration with Azure Functions

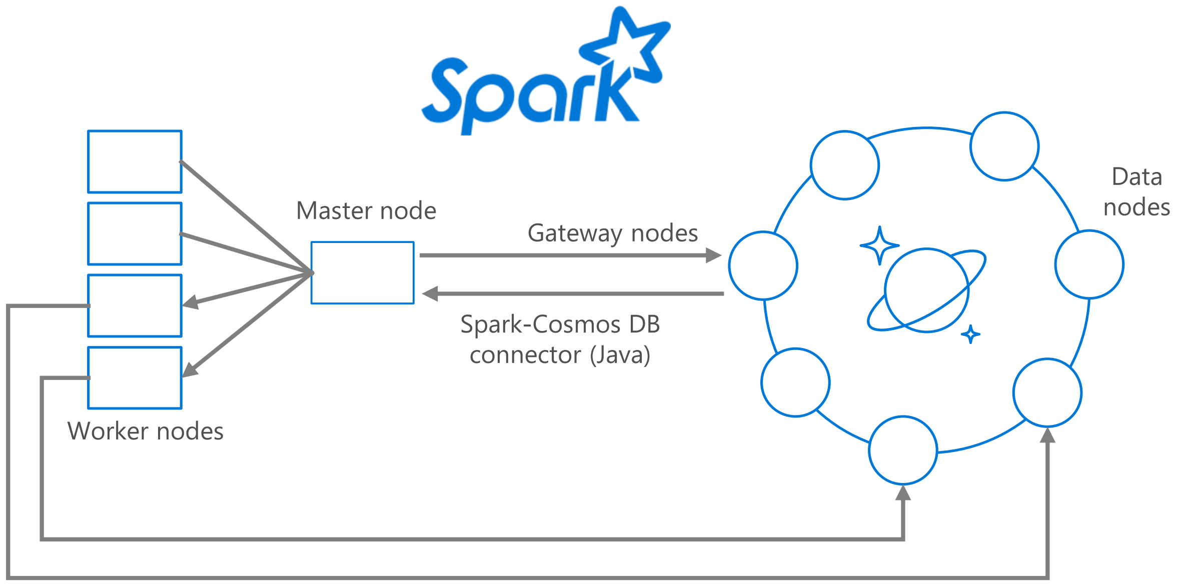


asos jet

RUN SPARK OVER OPERATIONAL DATA

Accelerate analysis of fast-changing, high-volume, global data.

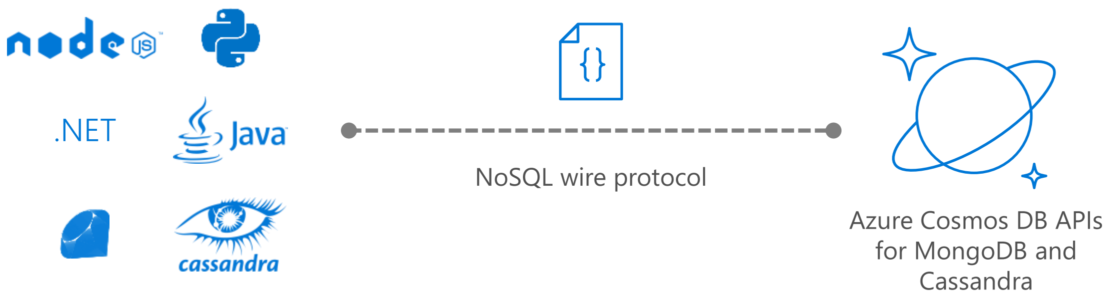
- Real-time big data processing across any data model
- Machine learning at scale over globally-distributed data
- Speeds analytical queries with automatic indexing and push-down predicate filtering
- Native integration with Spark Connector



LIFT AND SHIFT NOSQL APPS

Make data modernization easy with seamless lift and shift migration of NoSQL workloads to the cloud.

- Azure Cosmos DB APIs for MongoDB and Cassandra bring app data from anywhere to Azure Cosmos DB
- Leverage existing tools, drivers, and libraries, and continue using existing apps' current SDKs
- Turnkey geo-replication
- No infrastructure or VM management required

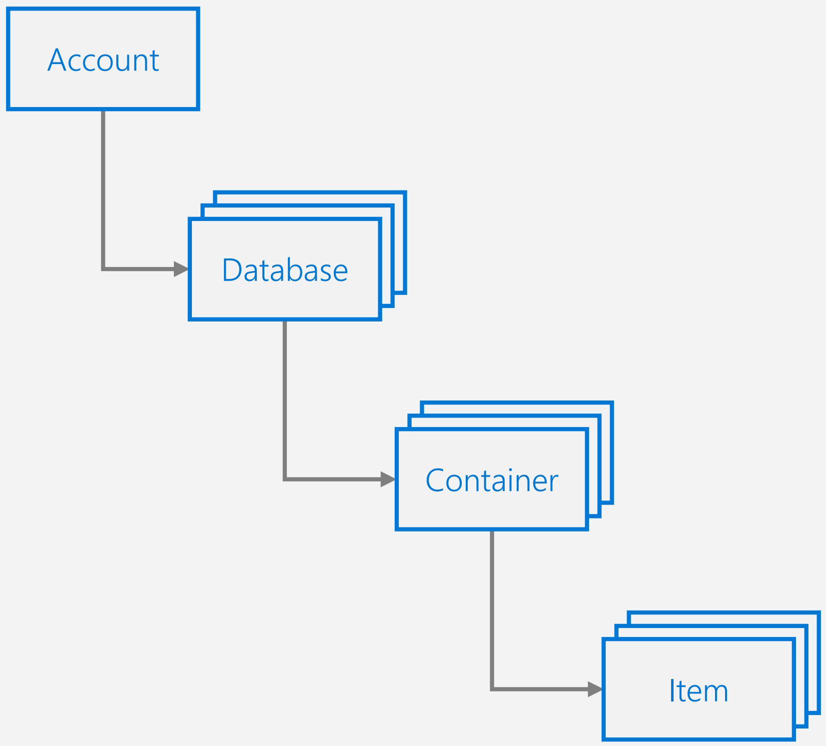


Resource Model

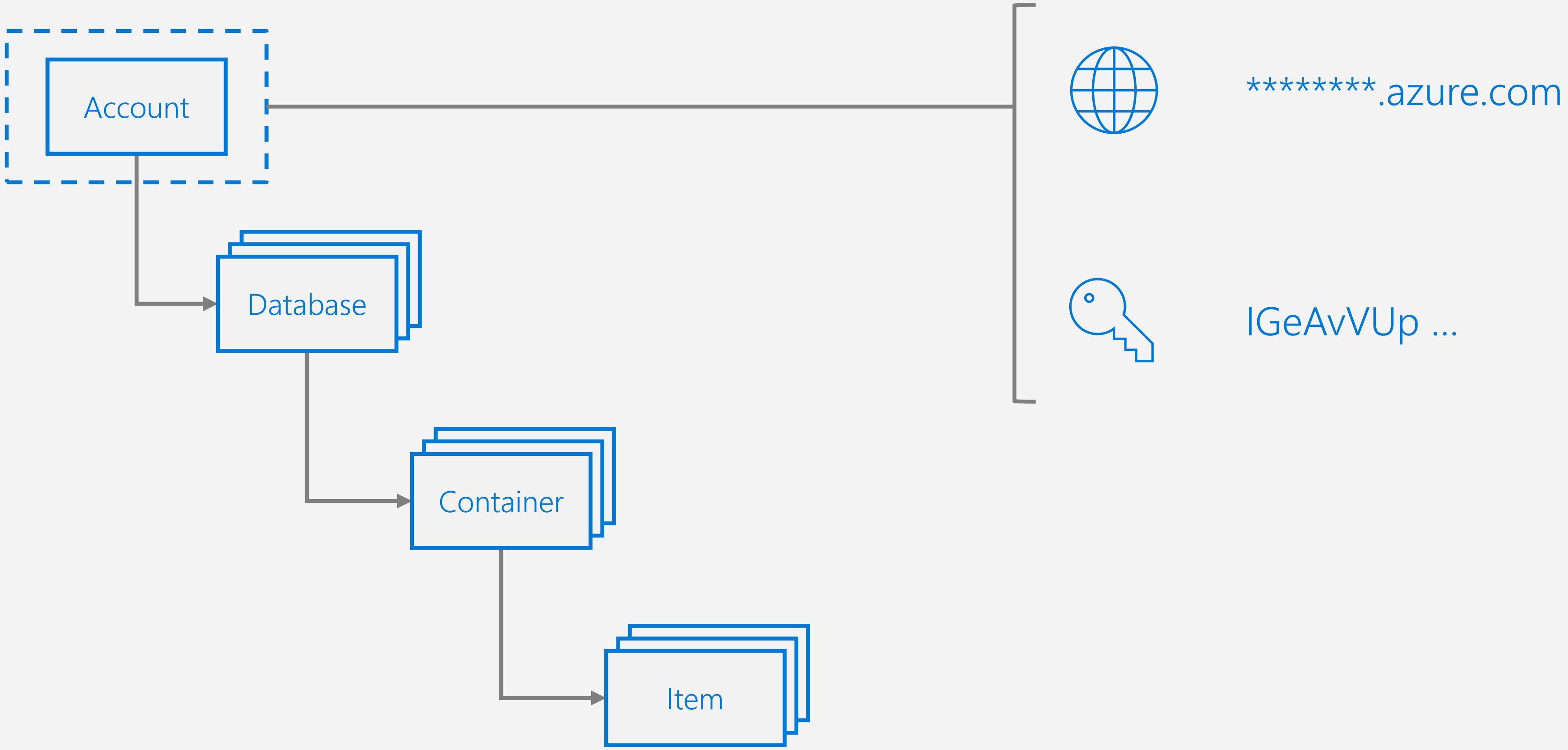
RESOURCE MODEL

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

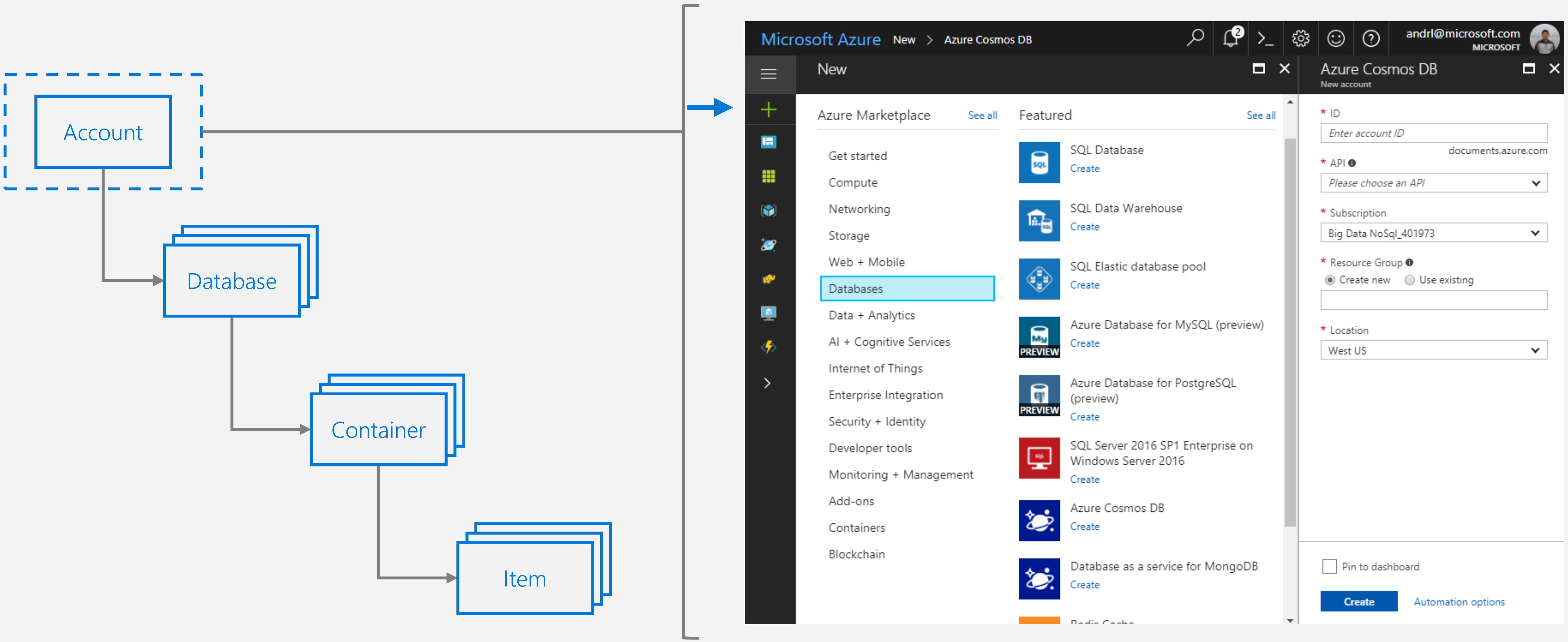
This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.



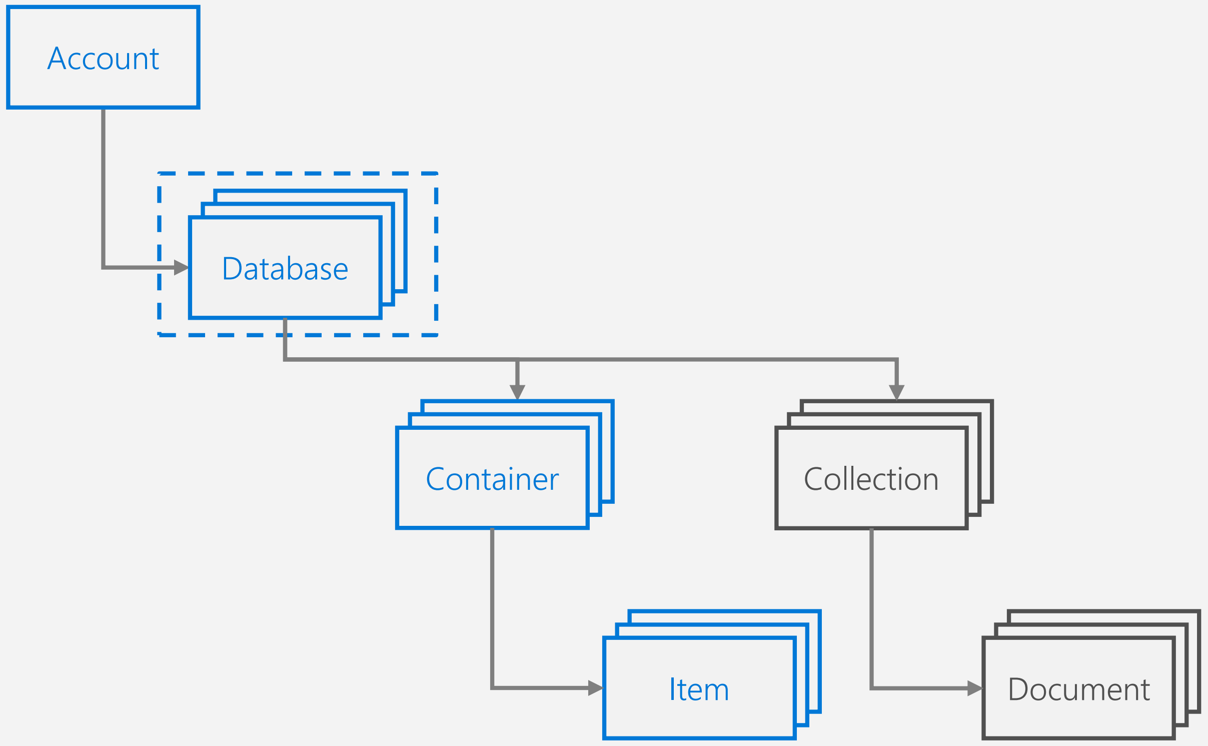
ACCOUNT URI AND CREDENTIALS



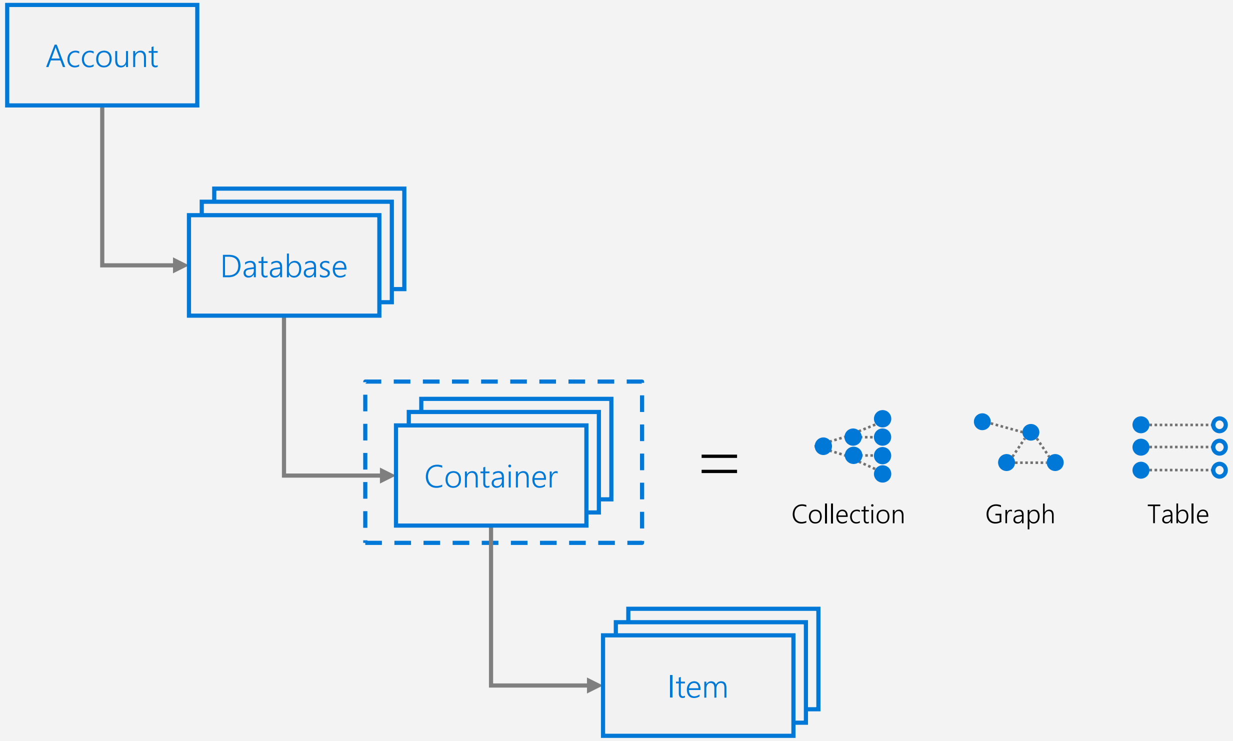
CREATING ACCOUNT



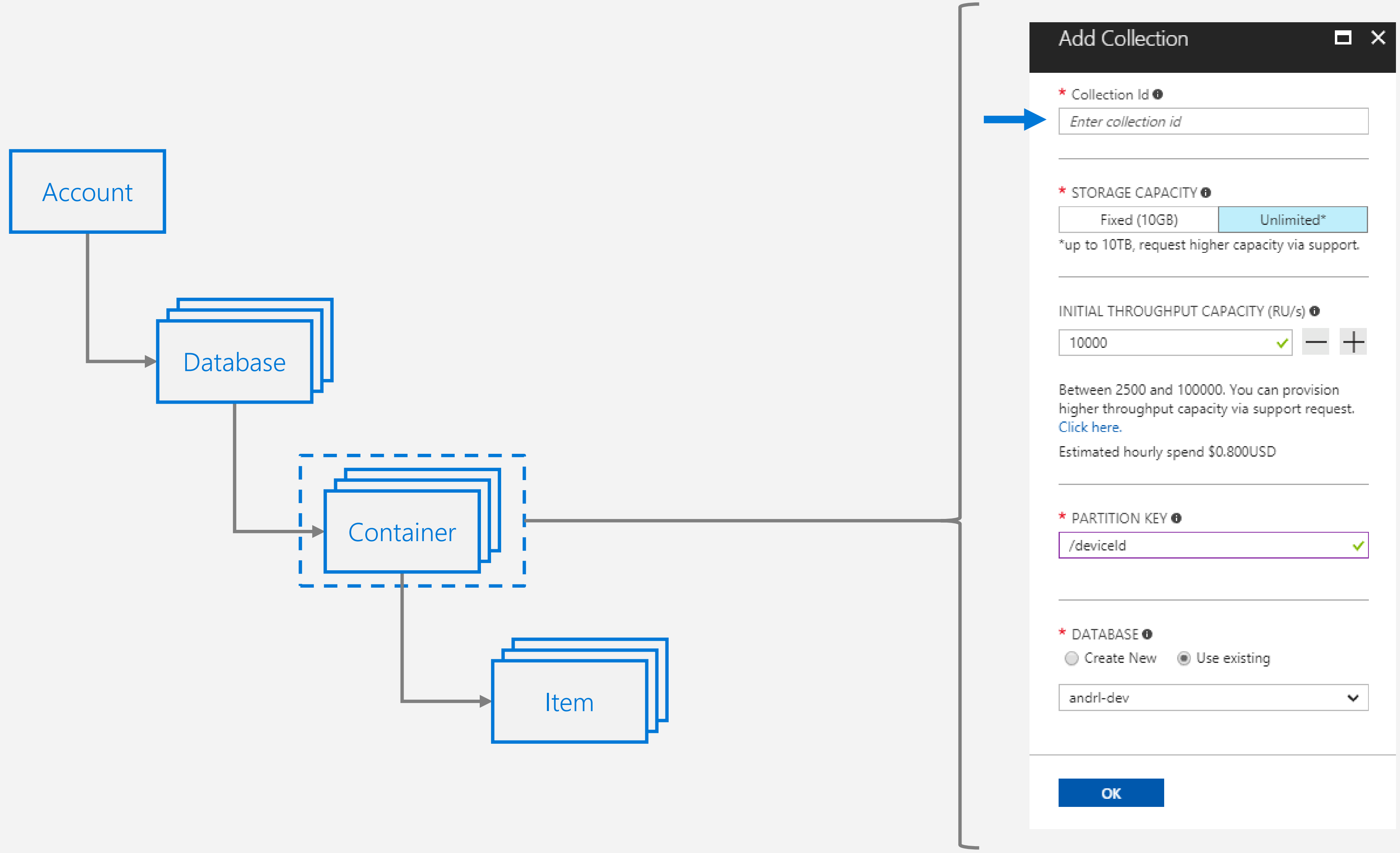
DATABASE REPRESENTATIONS



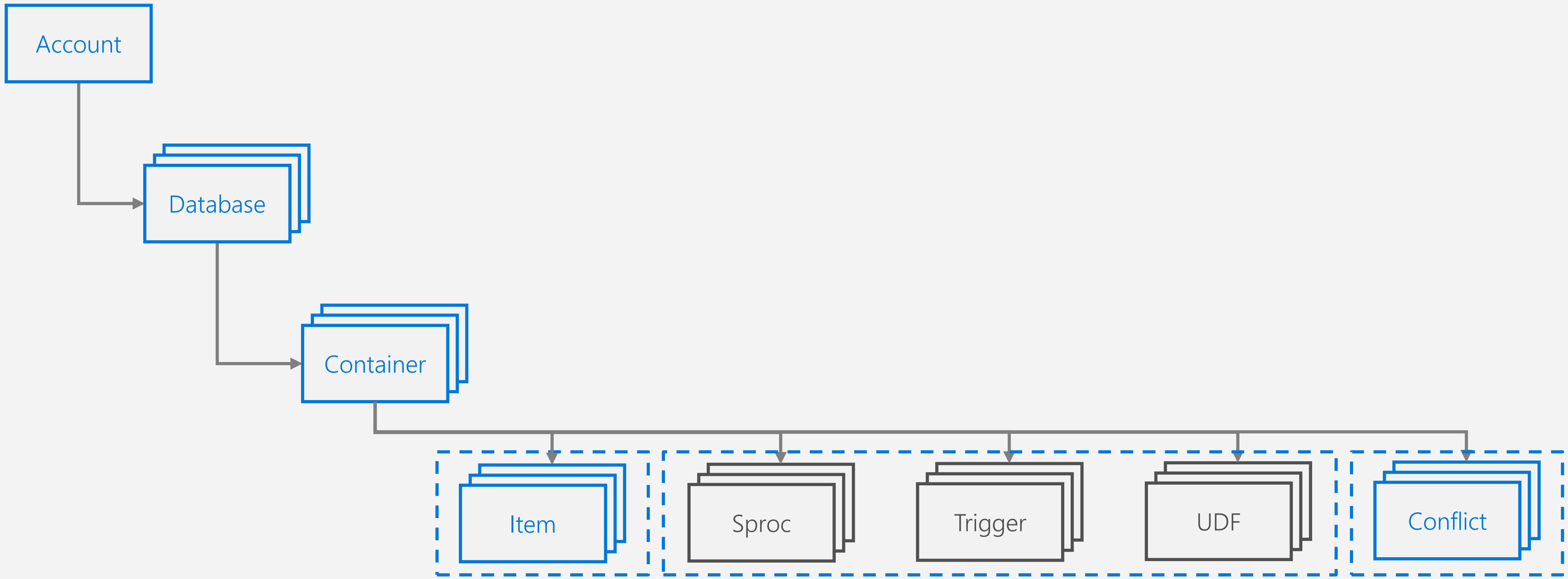
CONTAINER REPRESENTATIONS



CREATING COLLECTIONS – SQL API

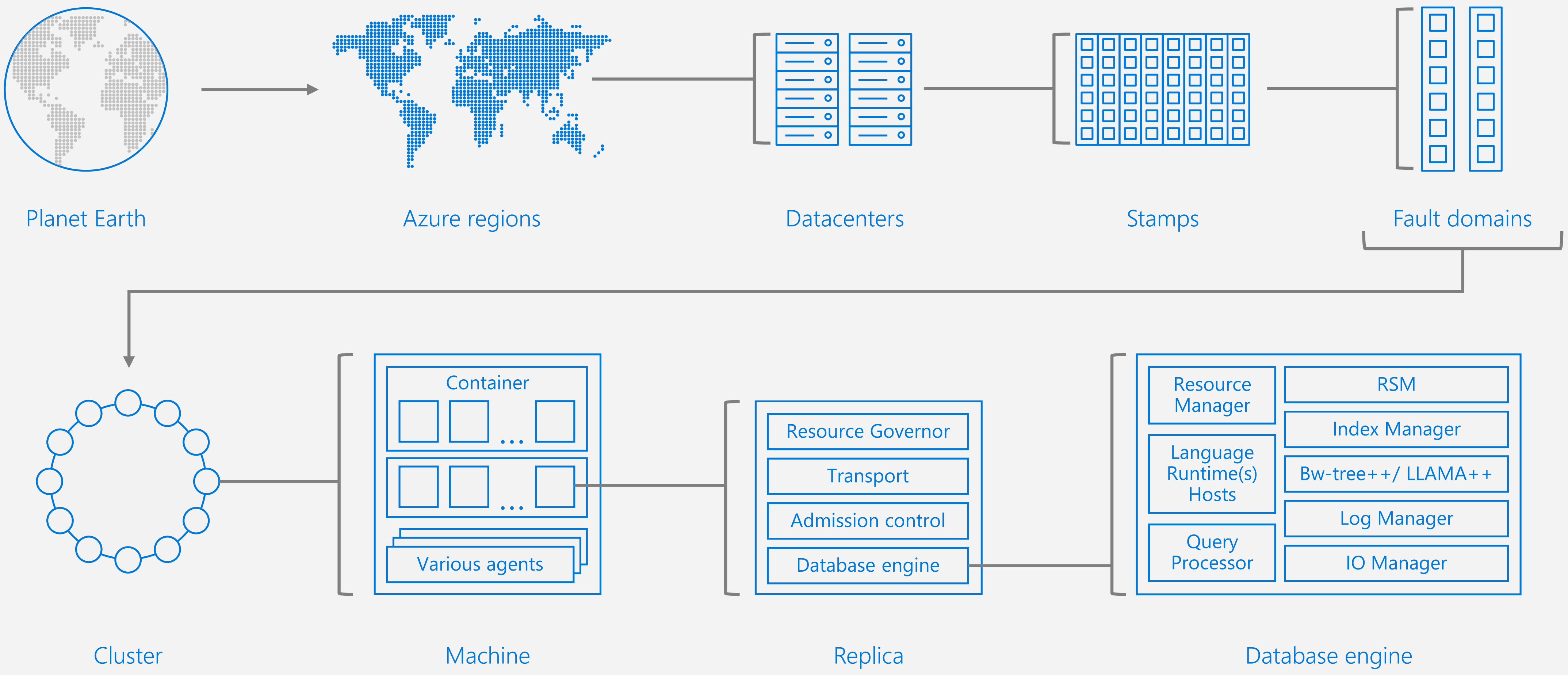


CONTAINER-LEVEL RESOURCES

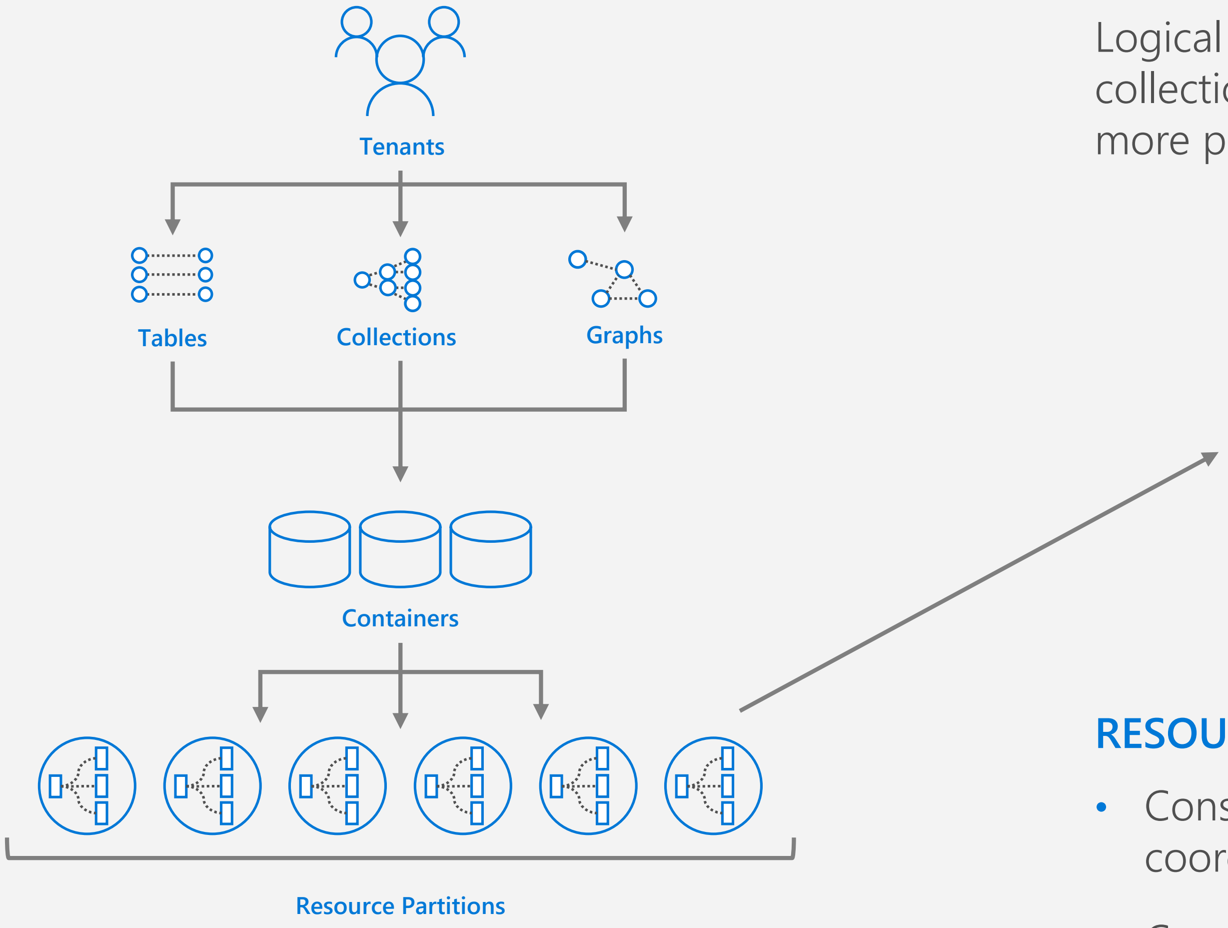


Partitioning

SYSTEM TOPOLOGY (BEHIND THE SCENES)

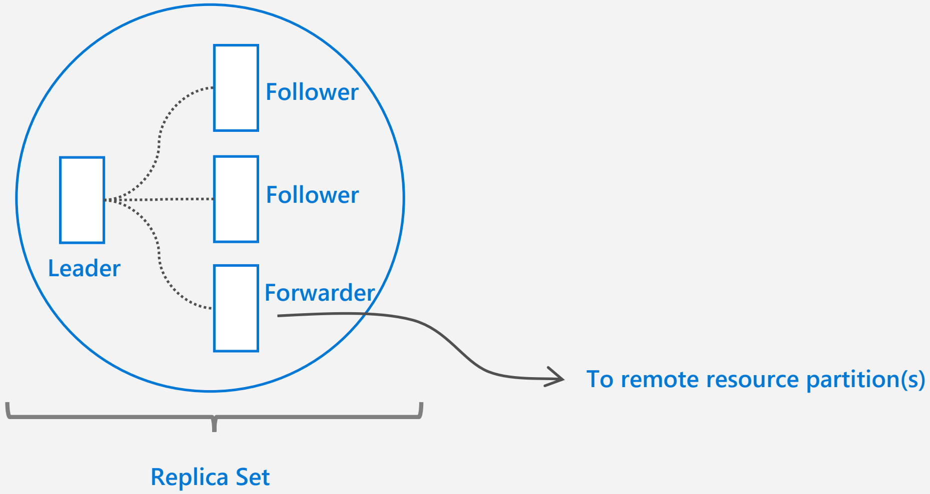


RESOURCE HIERARCHY



CONTAINERS

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



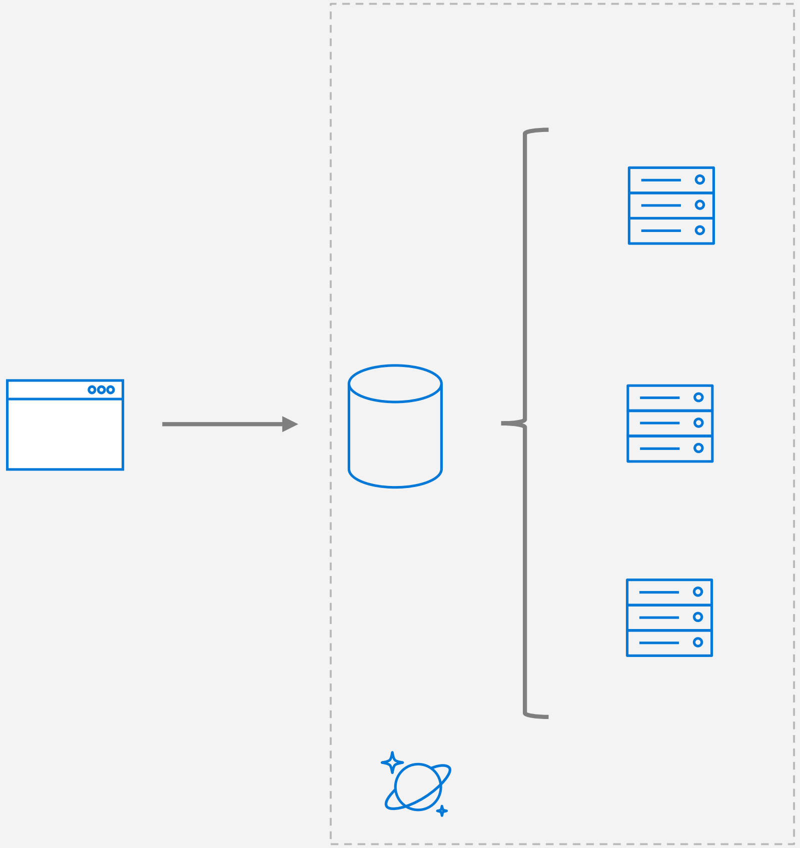
RESOURCE PARTITIONS

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

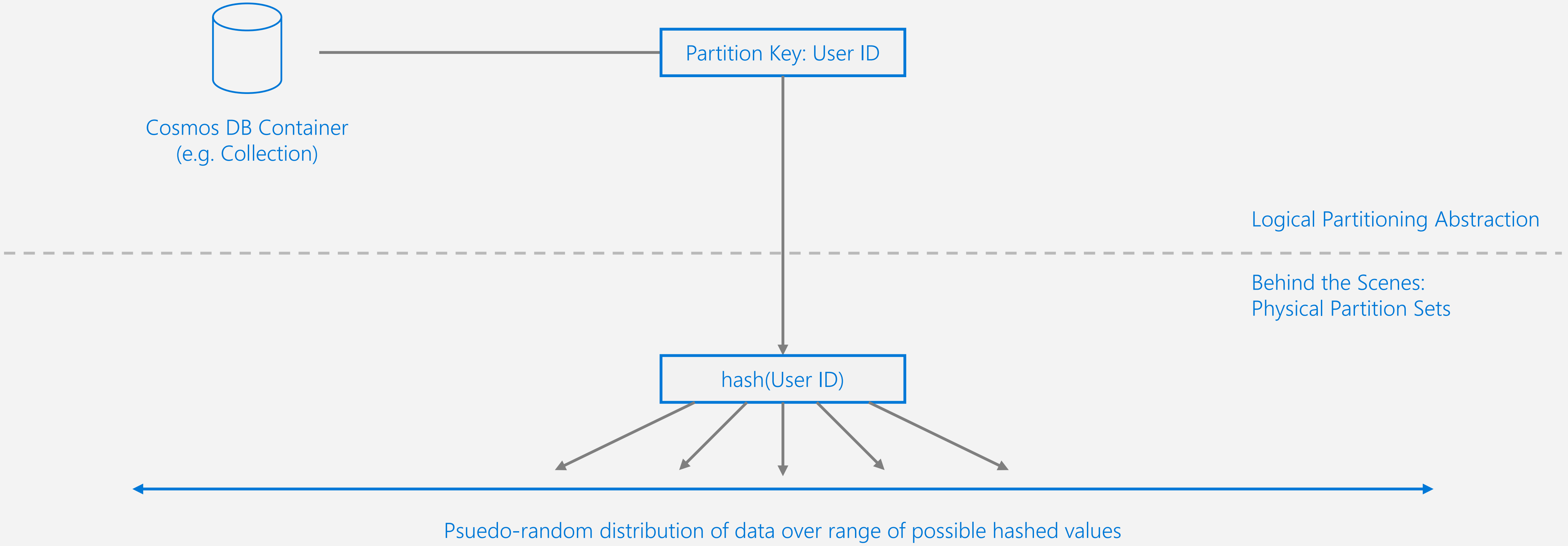
PARTITIONING

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.

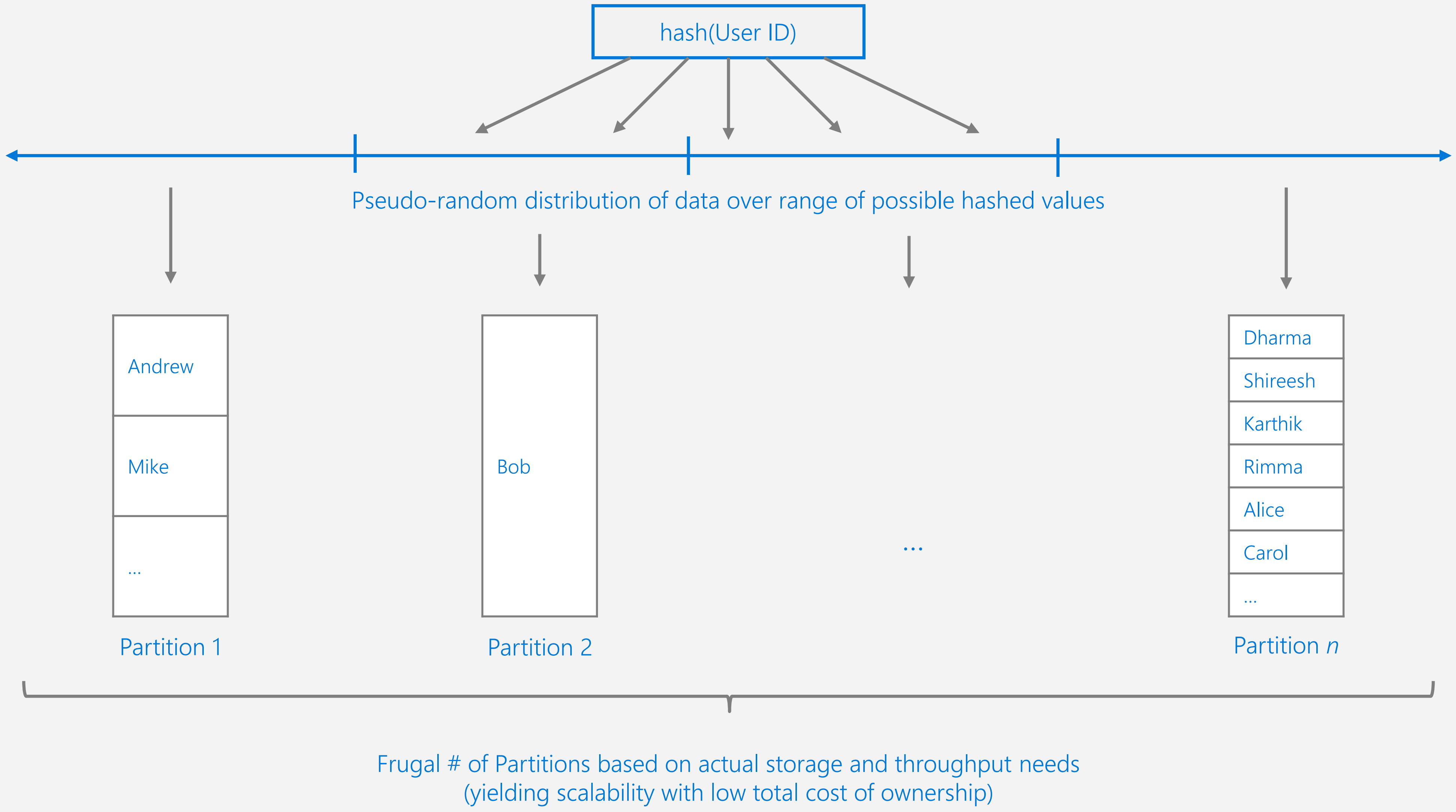


PARTITIONS



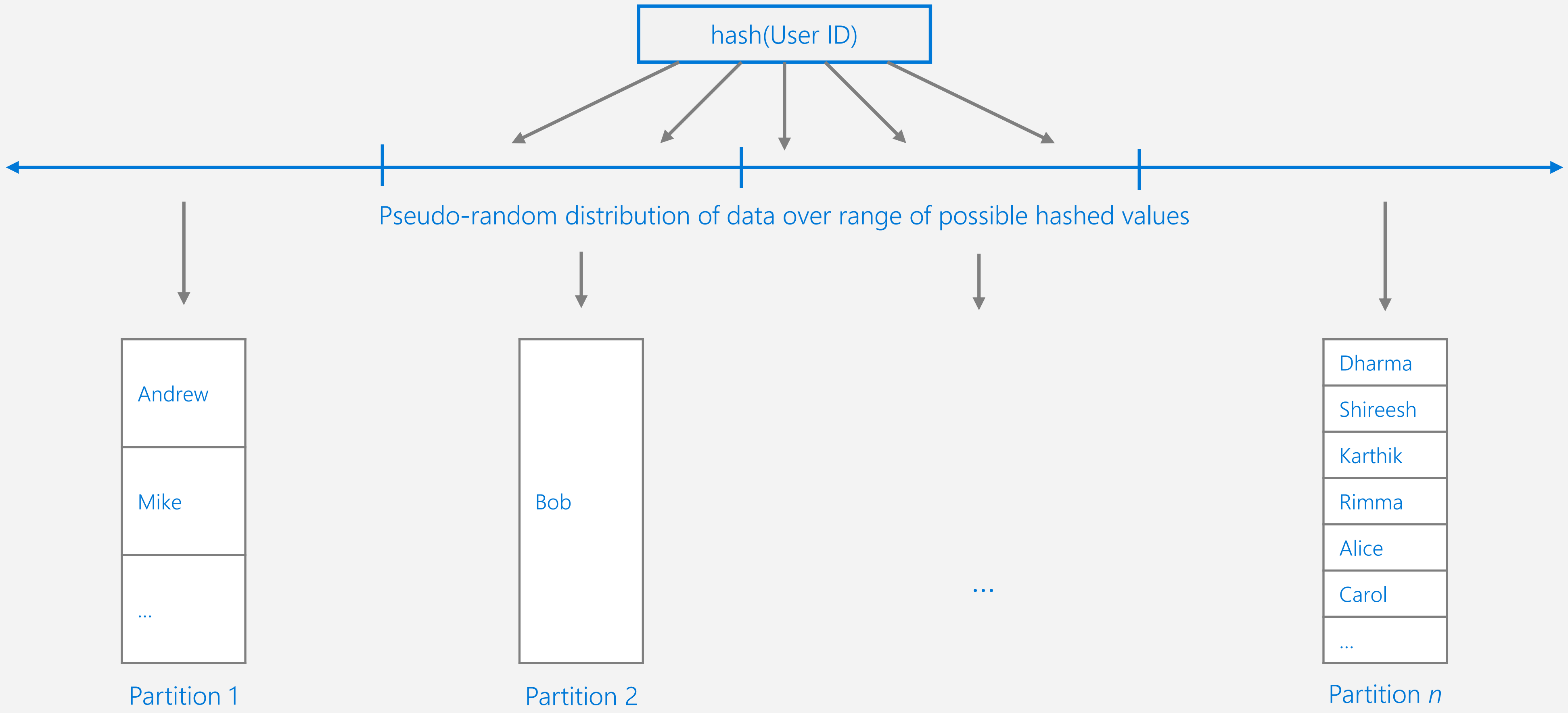
Partition Key: User ID
Cosmos DB Container
(e.g. Collection)
Logical Partitioning Abstraction
Behind the Scenes:
Physical Partition Sets
hash(User ID)
Psuedo-random distribution of data over range of possible hashed values

PARTITIONS



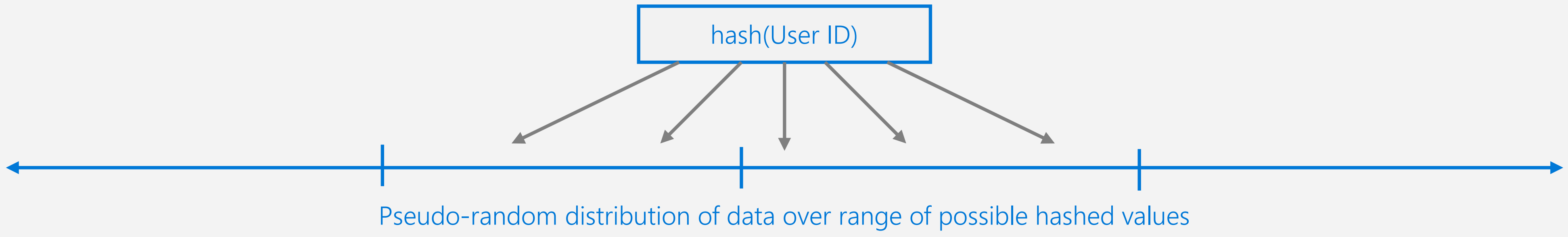
hash(User ID)
Pseudo-random distribution of data over range of possible hashed values
Andrew
Mike
...
Partition 1
Bob
Partition 2
...
Dharma
Shireesh
Karthik
Rimma
Alice
Carol
...
Partition n
Frugal # of Partitions based on actual storage and throughput needs
(yielding scalability with low total cost of ownership)

PARTITIONS



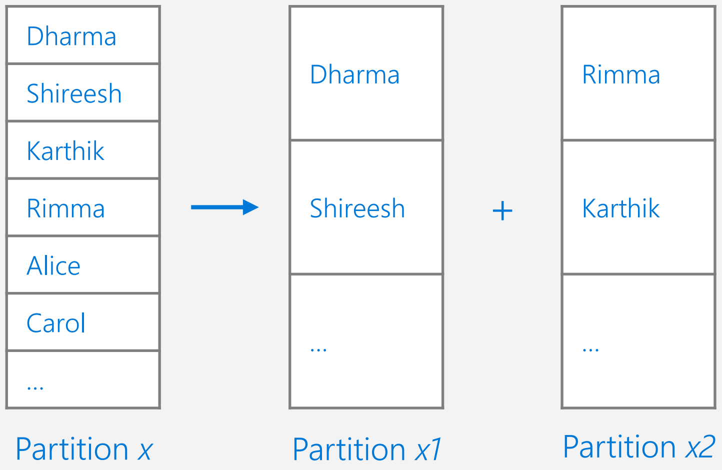
What happens when partitions need to grow?

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.



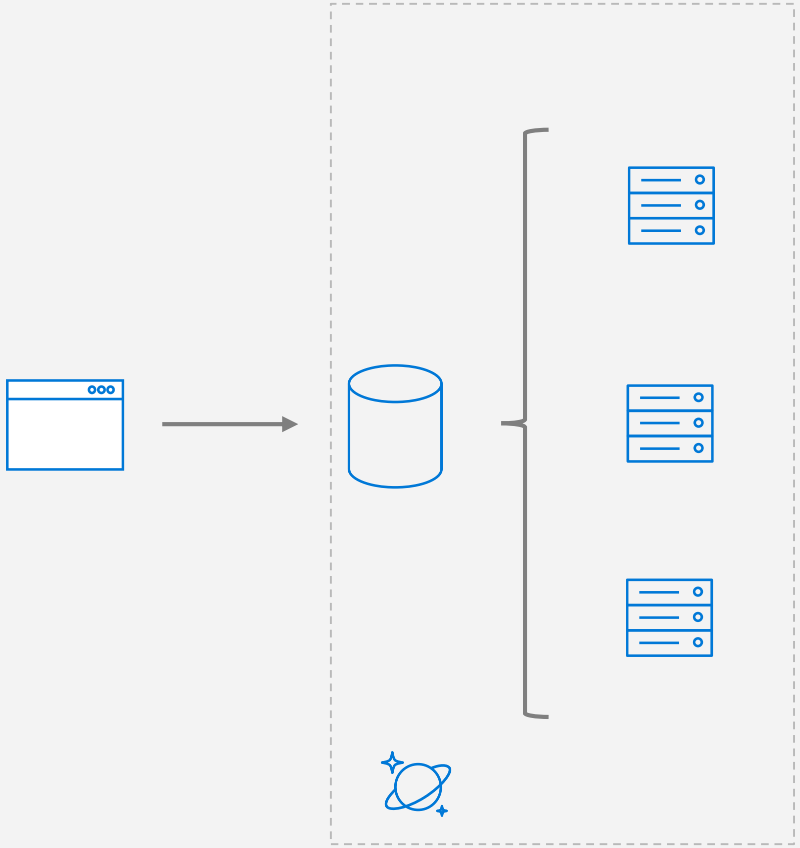
PARTITION DESIGN

IMPORTANT TO SELECT THE "RIGHT" PARTITION KEY

Partition keys acts as a **means for efficiently routing queries** and as a boundary for **multi-record** transactions.

KEY MOTIVATIONS

- Distribute Requests
- Distribute Storage
- Intelligently Route Queries for Efficiency



PARTITION DESIGN

EXAMPLE SCENARIO

Contoso Connected Car is a vehicle telematics company. They are planning to store vehicle telemetry data from millions of vehicles every second in Azure Cosmos DB to power predictive maintenance, fleet management, and driver risk analysis.

The partition key we select will be the scope for multi-record transactions.

WHAT ARE A FEW POTENTIAL PARTITION KEY CHOICES?

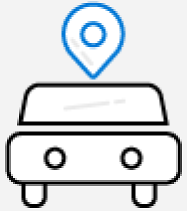
- Vehicle Model
- Current Time
- Device Id
- Composite Key – Device ID + Current Time

PARTITION KEY CHOICES

VEHICLE MODEL (e.g. Model A)

Most auto manufactures only have a couple dozen models. This will create a fixed number of logical partition key values; and is potentially the least granular option.

Depending how uniform sales are across various models – this introduces possibilities for hot partition keys on both storage and throughput.

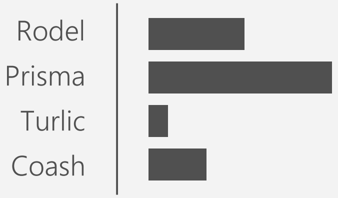


CURRENT MONTH (e.g. 2018-04)

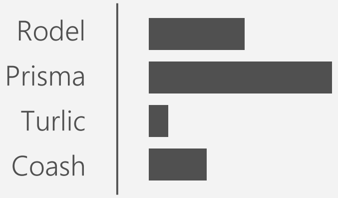
Auto manufacturers have transactions occurring throughout the year. This will create a more balanced distribution of storage across partition key values.

However, most business transactions occur on recent data creating the possibility of a hot partition key for the current month on throughput.

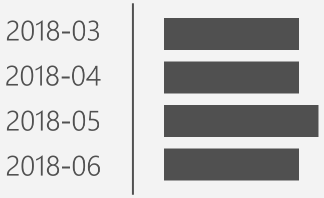
Storage Distribution



Throughput Distribution



Storage Distribution



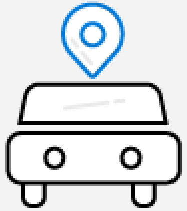
Throughput Distribution

PARTITION KEY CHOICES

DEVICE ID (e.g. Device123)

Each car would have a unique device ID. This creates a large number of partition key values and would have a significant amount of granularity.

Depending on how many transactions occur per vehicle, it is possible to a specific partition key that reaches the storage limit per partition key



COMPOSITE KEY (Device ID + Time)

This composite option increases the granularity of partition key values by combining the current month and a device ID. Specific partition key values have less of a risk of hitting storage limitations as they only relate to a single month of data for a specific vehicle.

Throughput in this example would be distributed more to logical partition key values for the current month.

Storage Distribution

Throughput Distribution

Storage Distribution

Throughput Distribution

PARTITION GRANULARITY

SELECT THE "RIGHT" LEVEL OF GRANULARITY FOR YOUR PARTITIONS

Partitions should be based on your most often occurring query and transactional needs. The goal is to **maximize granularity** and **minimize cross-partition requests**.

Don't be afraid to have more partitions!

More partition keys = More scalability

Example – Contoso Connected Car

PARTITION GRANULARITY

SELECT THE "RIGHT" LEVEL OF GRANULARITY FOR YOUR PARTITIONS

Consider storage & throughput thresholds

Consider cross-partition query likelihood

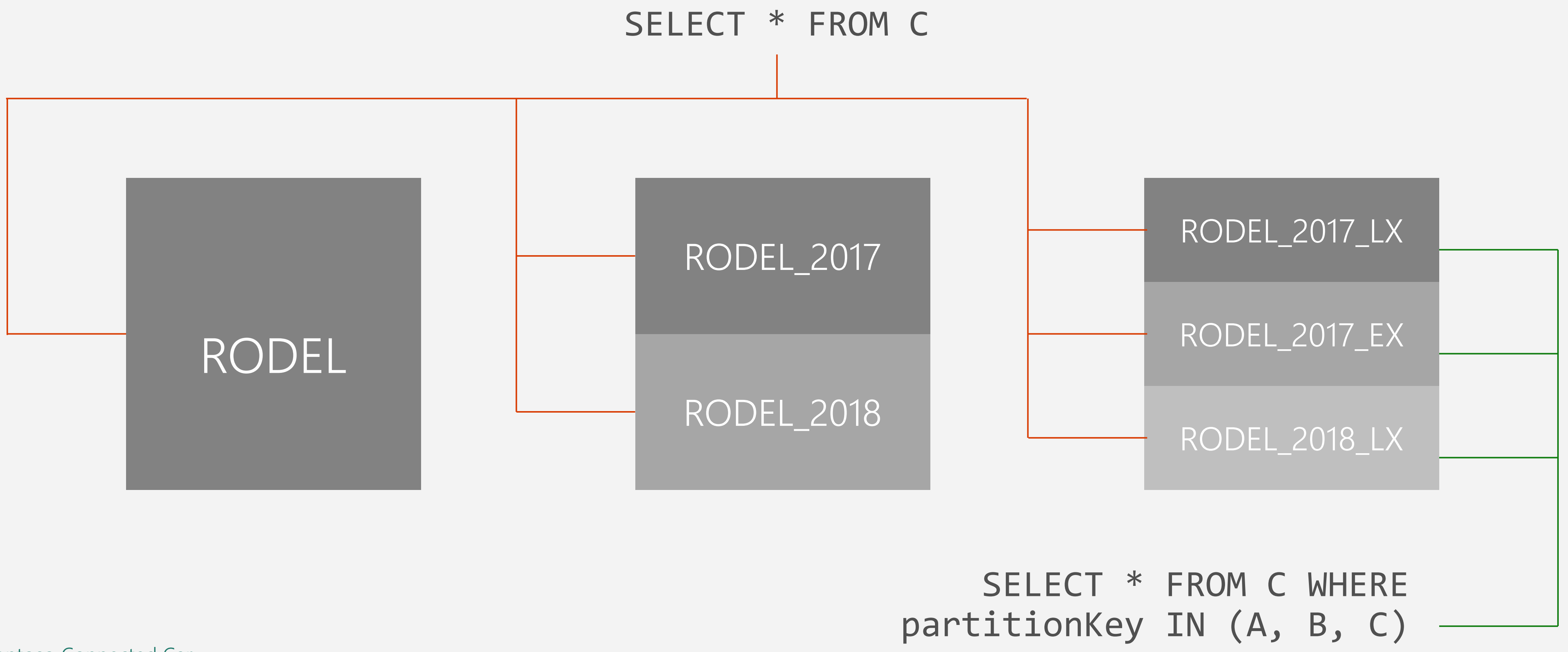
Don't be afraid to have more partitions!

More partition keys = More scalability

Example – Contoso Connected Car

PARTITION GRANULARITY

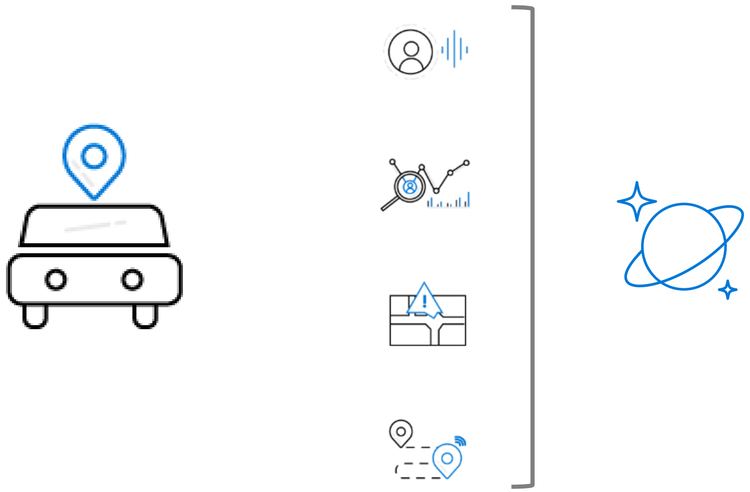
A CROSS-PARTITION QUERY IS NOT ALWAYS A BLIND FAN OUT QUERY

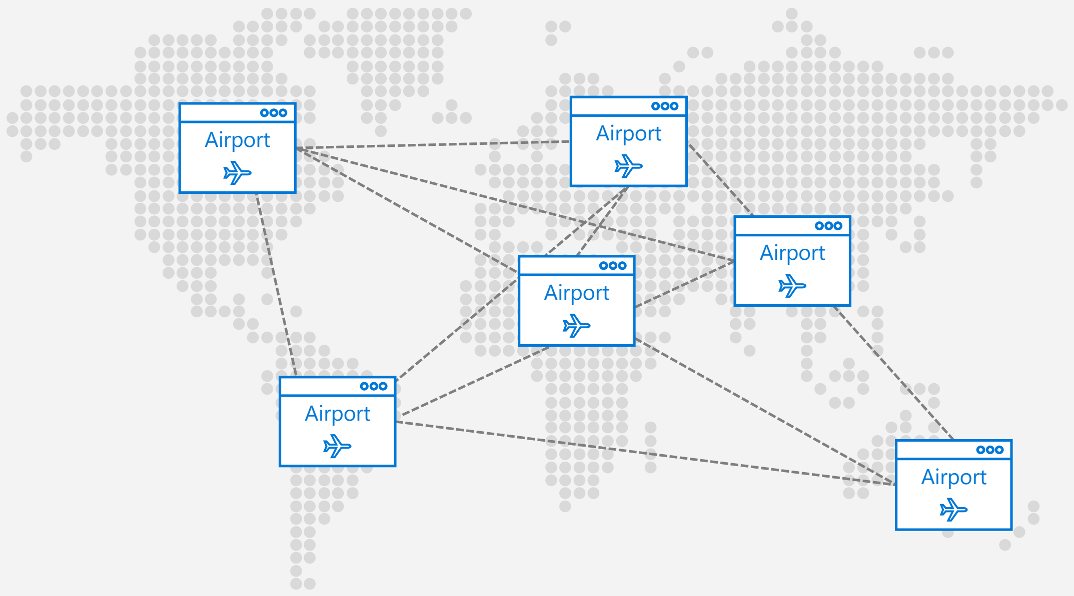


Example – Contoso Connected Car

PARTITION KEY SELECTION

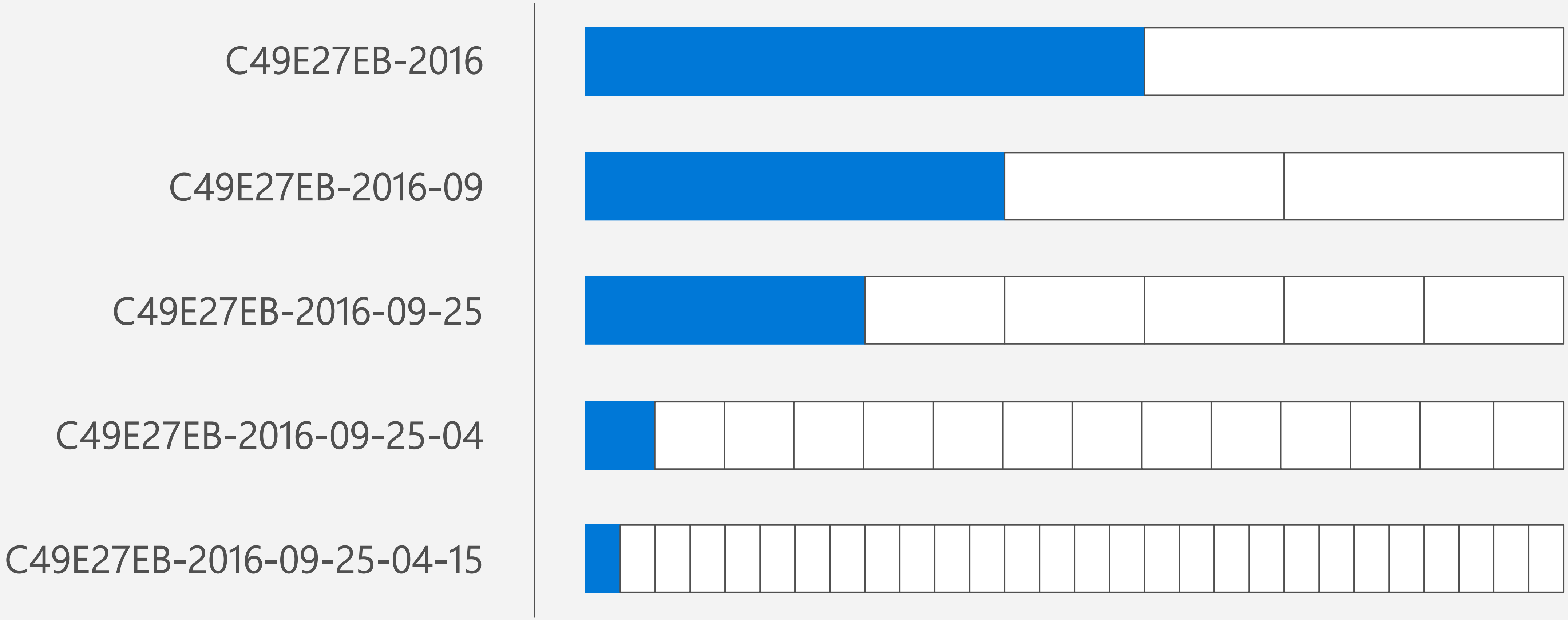
Contoso Connected Car is collecting and storing vehicle telemetry data from millions of vehicles. The team has decided to partition based on a composite key consisting of device id + current time when the interaction occurred.





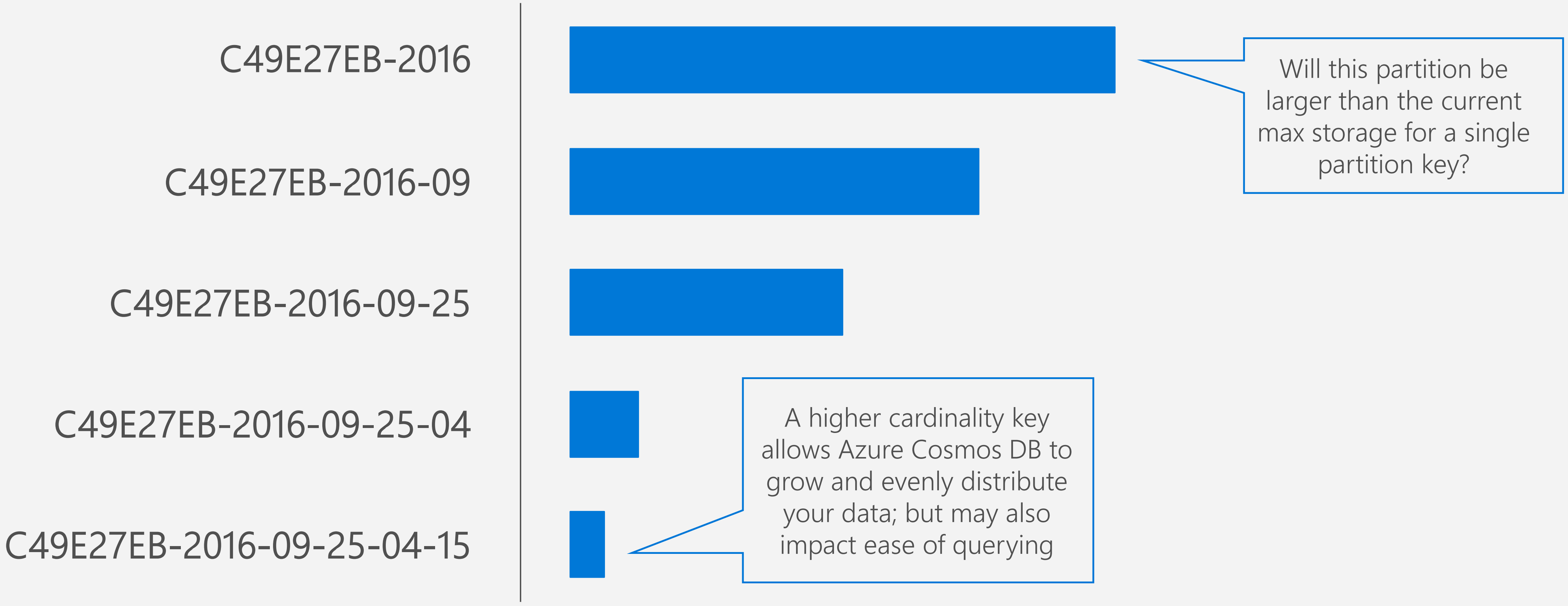
PARTITION KEY SCENARIO

Interaction that occurred on: **September 25, 2016 at 4:15 AM UTC**



PARTITION KEY SCENARIO

Interaction that occurred on: **September 25, 2016 at 4:15 AM UTC**



PARTITIONS

Best Practices: Design Goals for Choosing a Good Partition Key

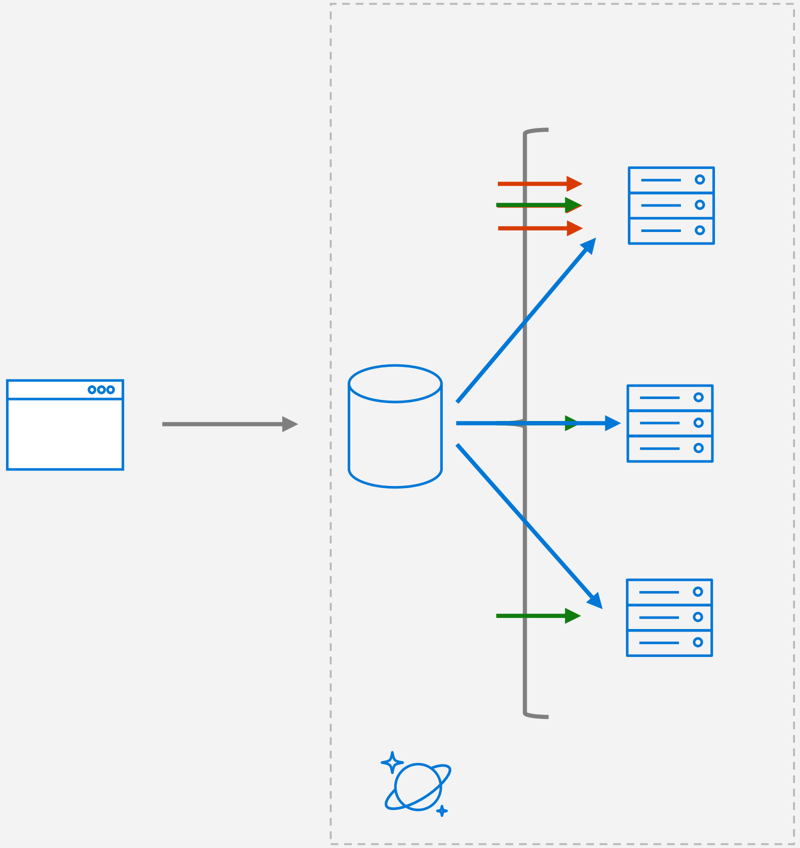
- Distribute the overall request + storage volume
    - Avoid “hot” partition keys
- Partition Key is scope for multi-record transactions and routing queries
    - Queries can be intelligently routed via partition key
    - Omitting partition key on query requires fan-out

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



PARTITION KEY STORAGE LIMITS

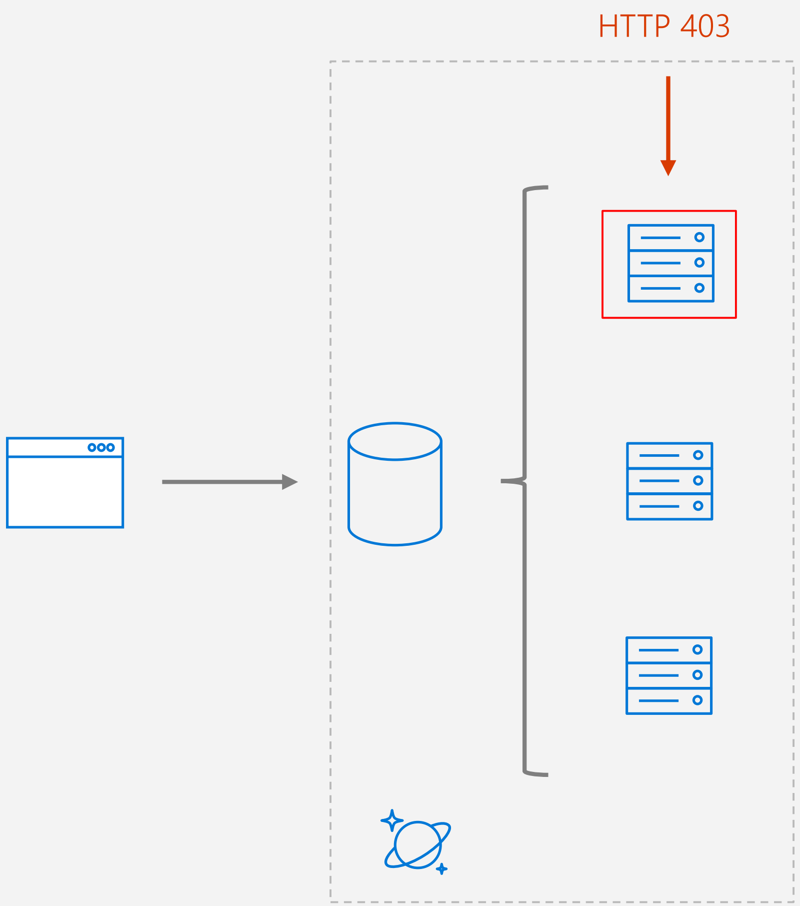
Containers support unlimited storage by dynamically allocating additional physical partitions

Storage for single partition key value (logical partition) is quota'ed to 10GB.

When a partition key reaches its provisioned storage limit, requests to create new resources will return a HTTP Status Code of 403 (Forbidden).

Azure Cosmos DB will automatically add partitions, and may also return a 403 if:

- An authorization token has expired
- A programmatic element (UDF, Stored Procedure, Trigger) has been flagged for repeated violations

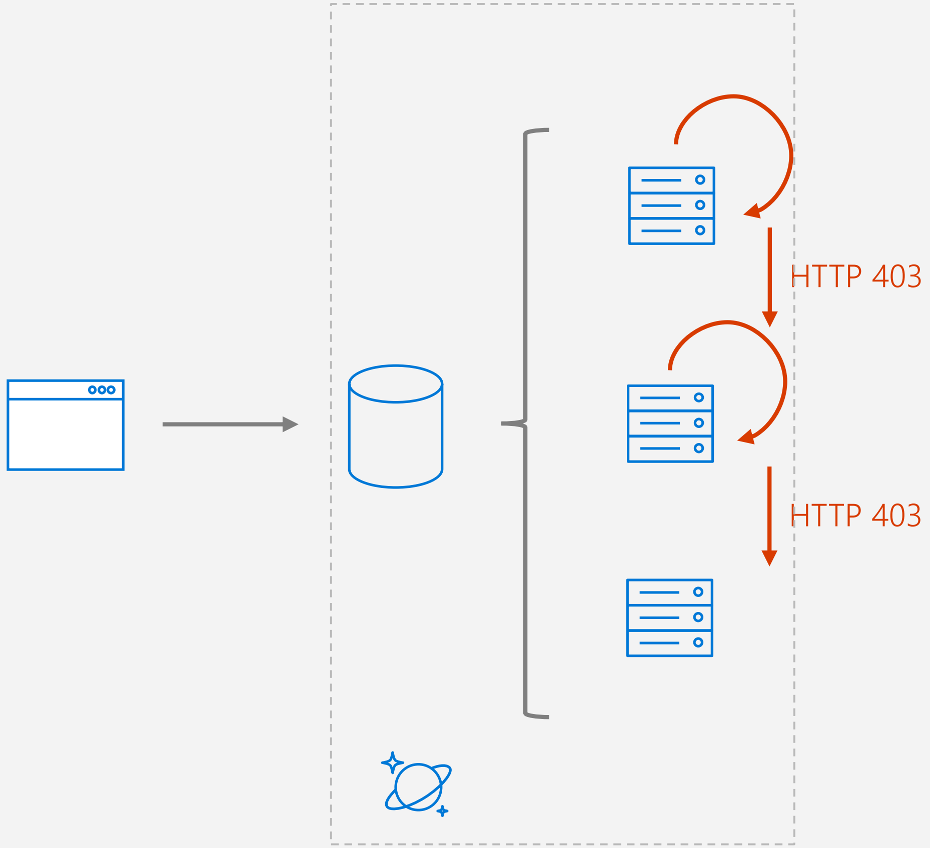


DESIGN PATTERNS FOR LARGE PARTITION KEYS

"LINKED LIST APPROACH" BY SREADING DATA ACROSS INCREMENTAL PARTITION KEY VALUES

For workloads that exceed quotas for a single partition key value, you can logically spread items across multiple partition keys within a container by using a suffix on the partition key value.

As a partition fills up, you can determine when to **increment** the partition key value by looking for the 403 status code in your application's logic.



DESIGN PATTERNS FOR LARGE PARTITION KEYS

"CIRCULAR BUFFER" APPROACH BY REUSING UNIQUE IDS

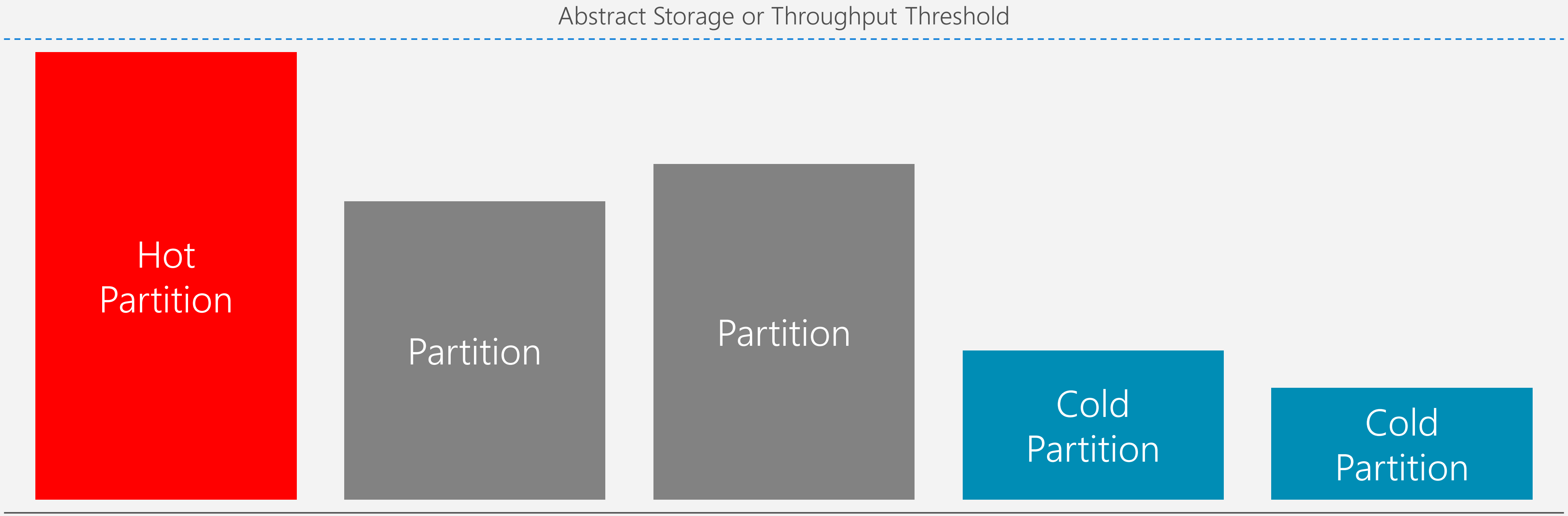
As you insert new items into a container's partition, you can increment the unique id for each item in the partition.

When you get a 403 status code, indicating the partition is full, you can restart your unique id and upsert the items to replace older documents.

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**.



QUERY FAN-OUT

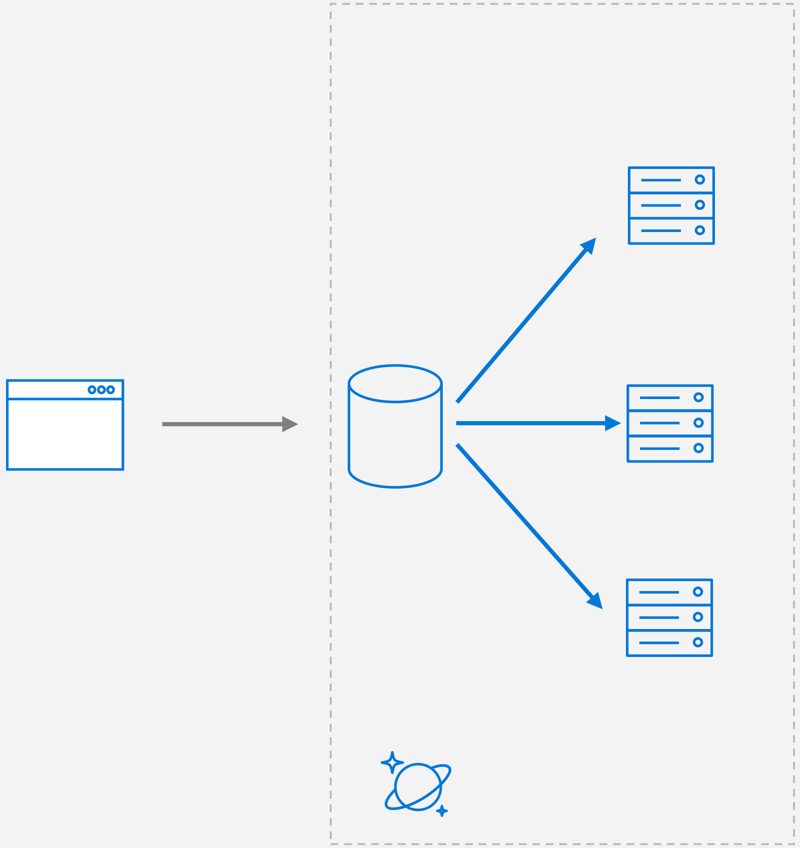
CROSS-PARTITION QUERIES CAN BE PERFORMED SERVER-SIDE OR CLIENT-SIDE

Cross-partition queries are opt-in

- Cross-partition queries can be tuned and parallelized

Creates a bottleneck

- Must wait for all partitions to return before the query is "done"



CROSS-PARTITION SDK EXAMPLE

```
IQueryable<DeviceReading> crossPartitionQuery = client.CreateDocumentQuery<DeviceReading>(
    UriFactory.CreateDocumentCollectionUri("db", "coll"),
    new FeedOptions {
          EnableCrossPartitionQuery = true,
          MaxDegreeOfParallelism = 10,
          MaxBufferedItemCount = 100
    })
    .Where(m => m.MetricType == "Temperature" && m.MetricValue > 100)
    .OrderBy(m => m.MetricValue);
```

.NET

CROSS-PARTITION SDK EXAMPLE

```
var querySpec = {
         query: 'SELECT * FROM container c'
};

var feedOptions = {
         enableCrossPartitionQuery = true
         maxDegreeOfParallelism = 10
};

client.queryDocuments(collectionLink, querySpec, feedOptions)
         .toArray(function (err, results) {
         }
```

CROSS-PARTITION SDK EXAMPLE

```java
FeedOptions options = new FeedOptions();
options.setEnableCrossPartitionQuery(true);

Iterator<Document> it = client.queryDocuments(
    collectionLink,
    "SELECT * from r",
    options
).getQueryIterator();
```

QUERY FAN OUT

QUERYING ACROSS PARTITIONS IS NOT ALWAYS A BAD THING

If you have **relevant data to return**, creating a cross-partition query is a perfectly acceptable workload with a predictable throughput.

In an ideal situation, queries are **filtered to only include relevant partitions**.

BLIND QUERY FAN-OUTS CAN ADD UP

You are charged **~1 RU** for each partition that doesn't have any relevant data.

Multiple fan-out queries can quickly max out RU/s for each partition

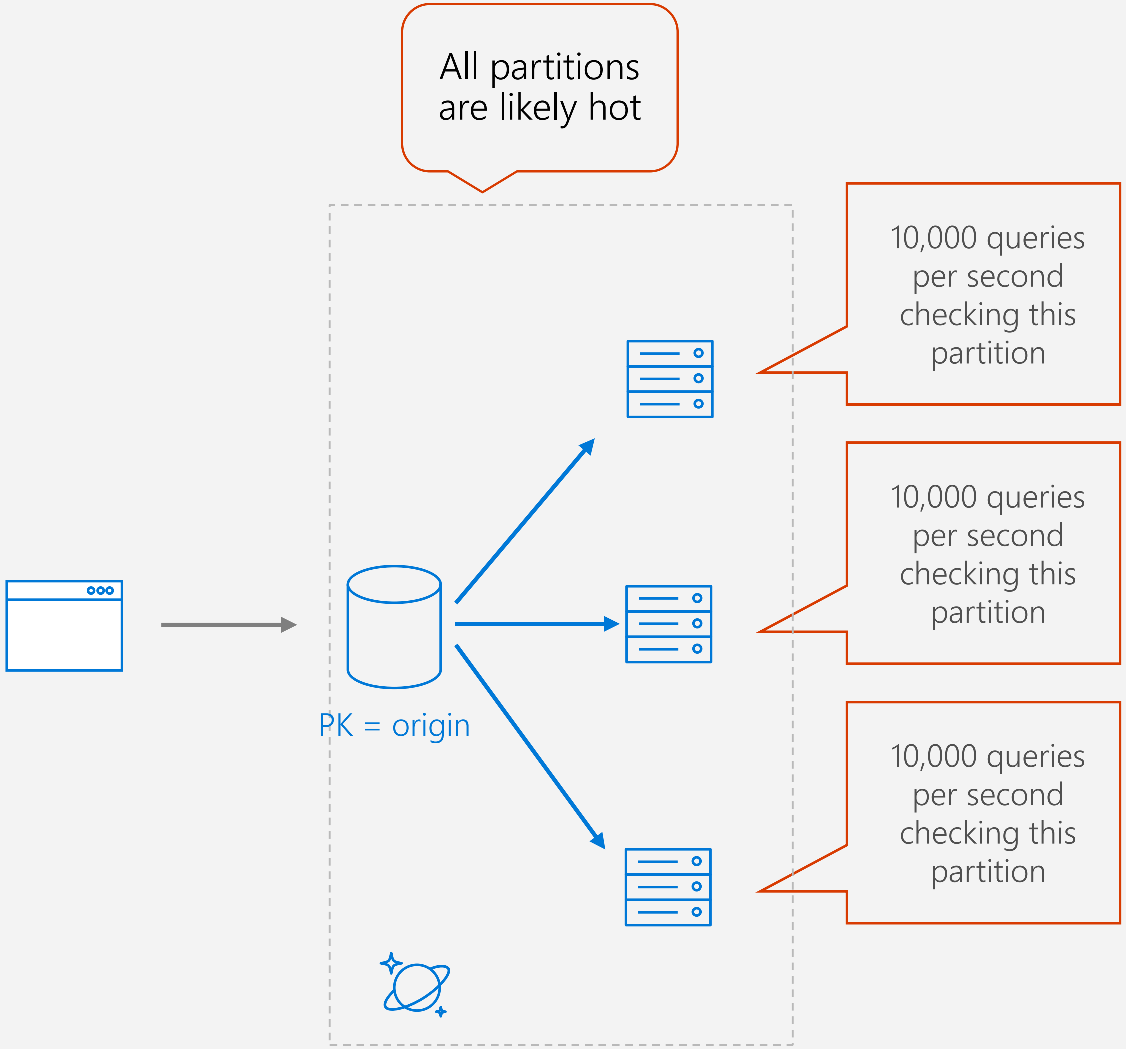
QUERY FAN OUT

CONCURRENCY AND FAN-OUT QUERIES

>**10,000** fan-out queries in a second can leave all partitions hot

Example: Query on a vehicle database, partitioned by model name, where the query is filtering by year without **filtering to only include relevant partitions**.

```
SELECT * FROM car a WHERE a.year = "2015"

>10,000 more queries
per second

SELECT * FROM car a WHERE a.year = "2016"
```



QUERY FAN OUT

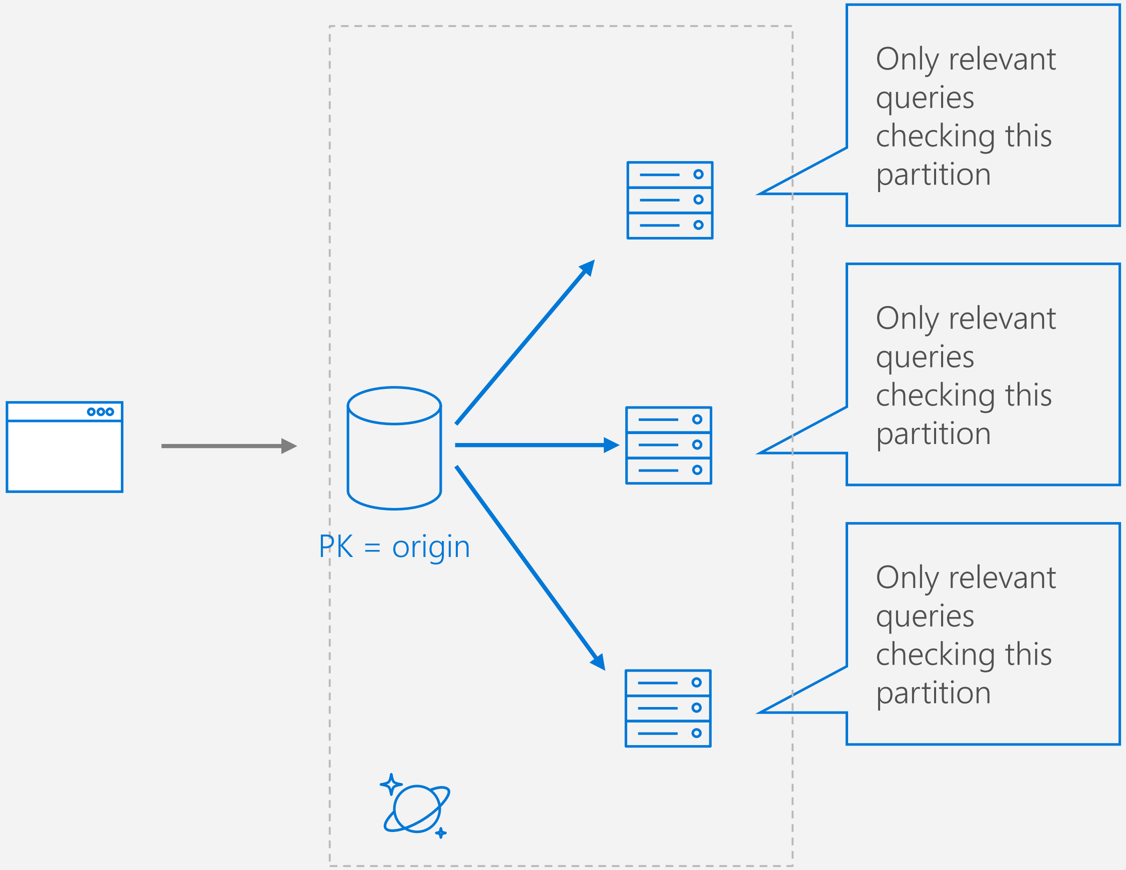
CONCURRENCY AND FAN-OUT QUERIES

Filtering queries to only include relevant partition key values **reduces the amount of wasted effort** and focuses queries on those relevant partitions.

```
SELECT * FROM car a
        WHERE a.model = “TURLIC” AND a.year = “2015”
```

>10,000 more queries per second

```
SELECT * FROM car a
        WHERE a.model = “COASH” AND a.year = “2016”
```

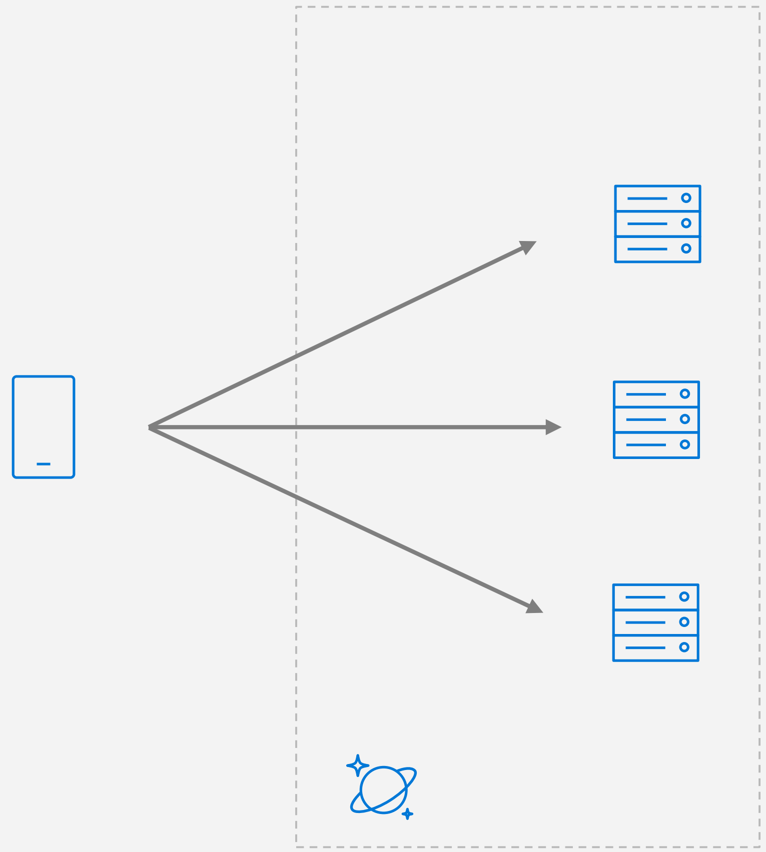


Example – Contoso Connected Car

CLIENT-SIDE QUERY FAN-OUT

USING CLIENT-SIDE THREADS AND PARRALELISM TO FAN-OUT QUERIES ACROSS PARTITIONS

- Leverage existing client device power
- Perform queries in parallel across units
- Potentially process early-finish results before parallel requests are complete

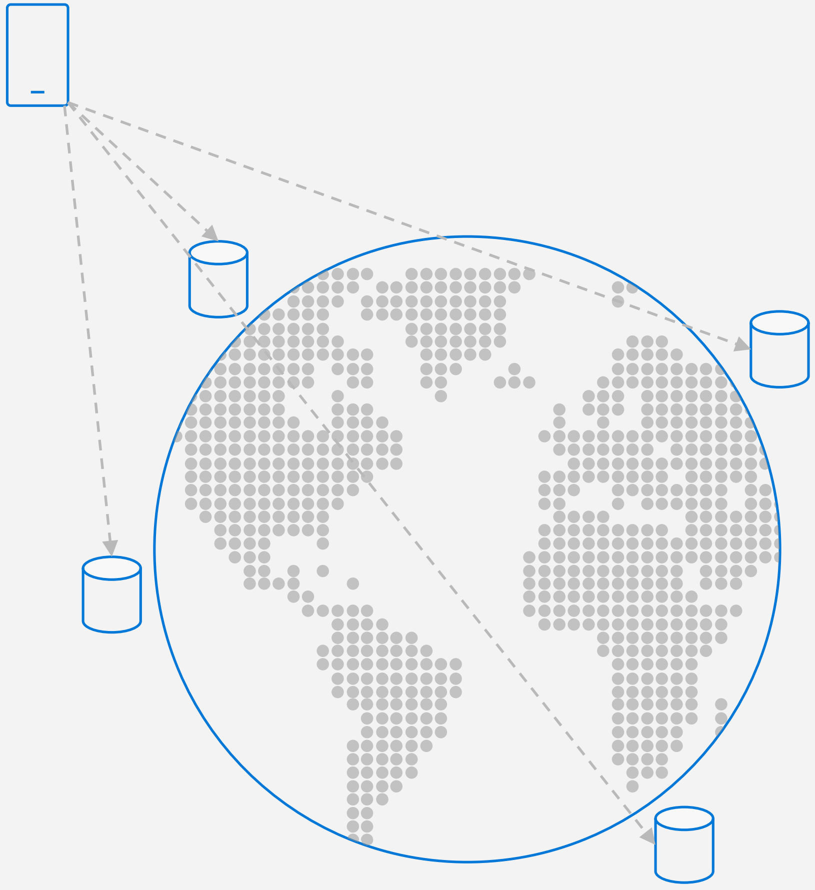


Querying

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

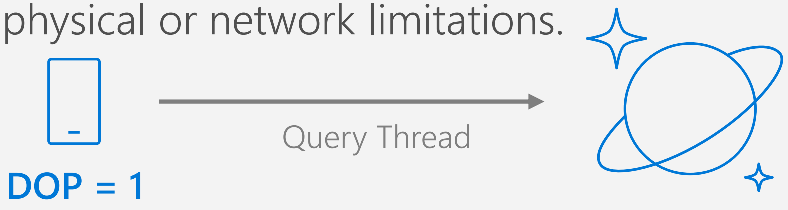
Analyze the query execution metrics to identify potential rewrites of query and data shapes

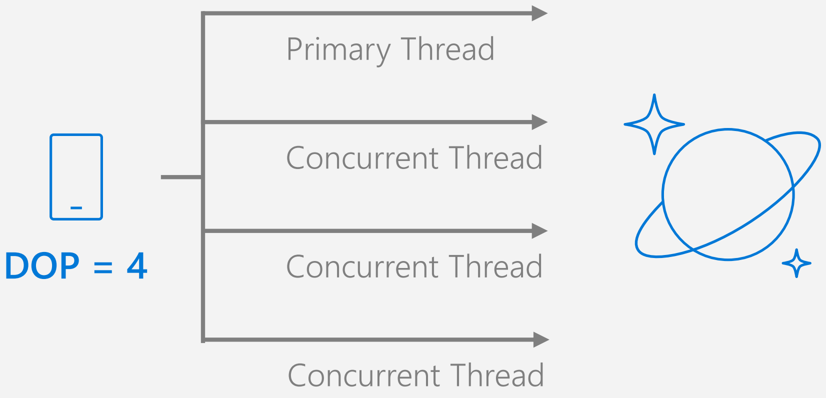
CLIENT QUERY PARALLELISM

CROSS-PARTITOIN QUERIES CAN BE PARRALELIZED TO USE AS MANY THREADS AS POSSIBLE

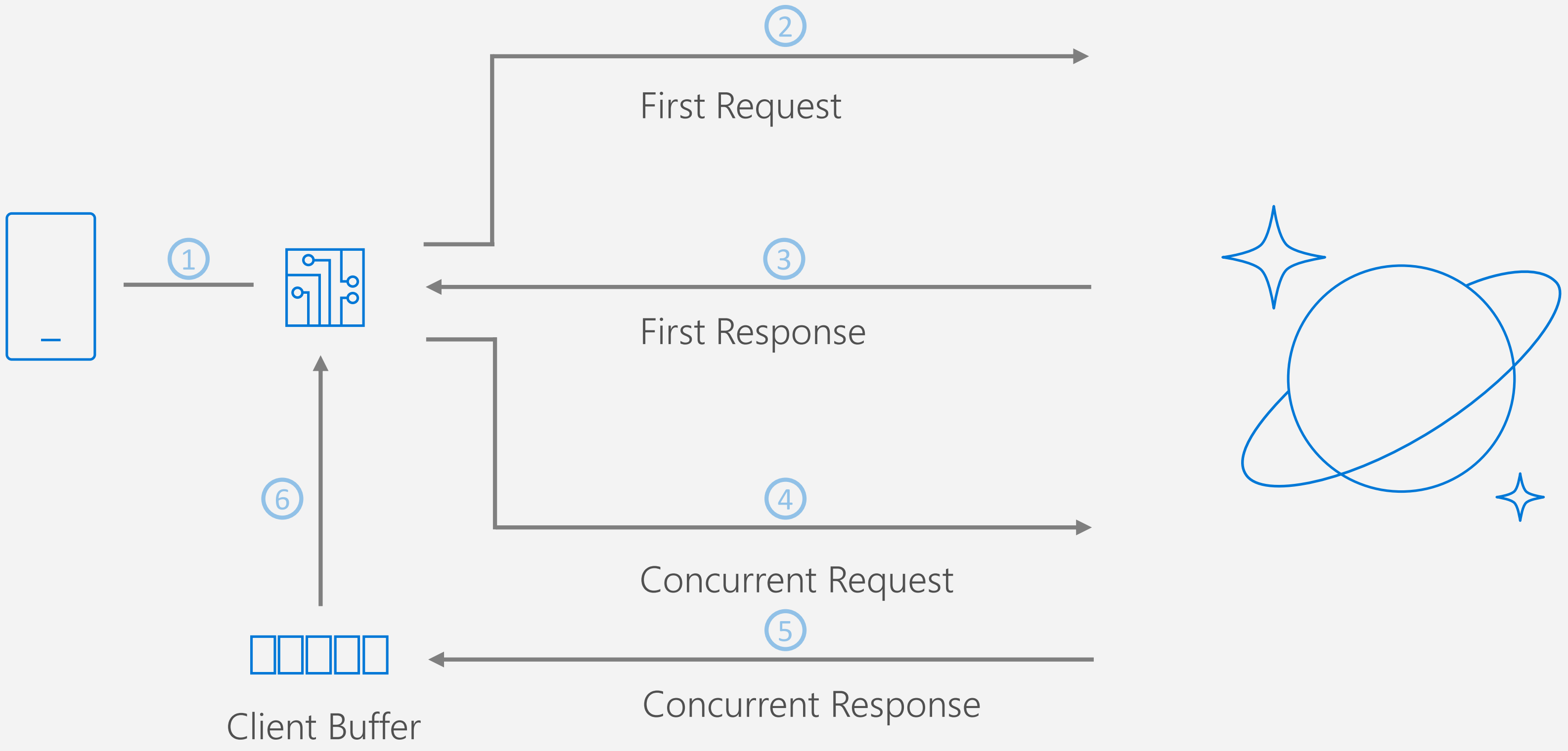
Modern processors ship with both physical and virtual (hyper-threading) cores. For any given cross-partition query, the SDK can use concurrent threads to issue the query across the underlying partitions.

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.



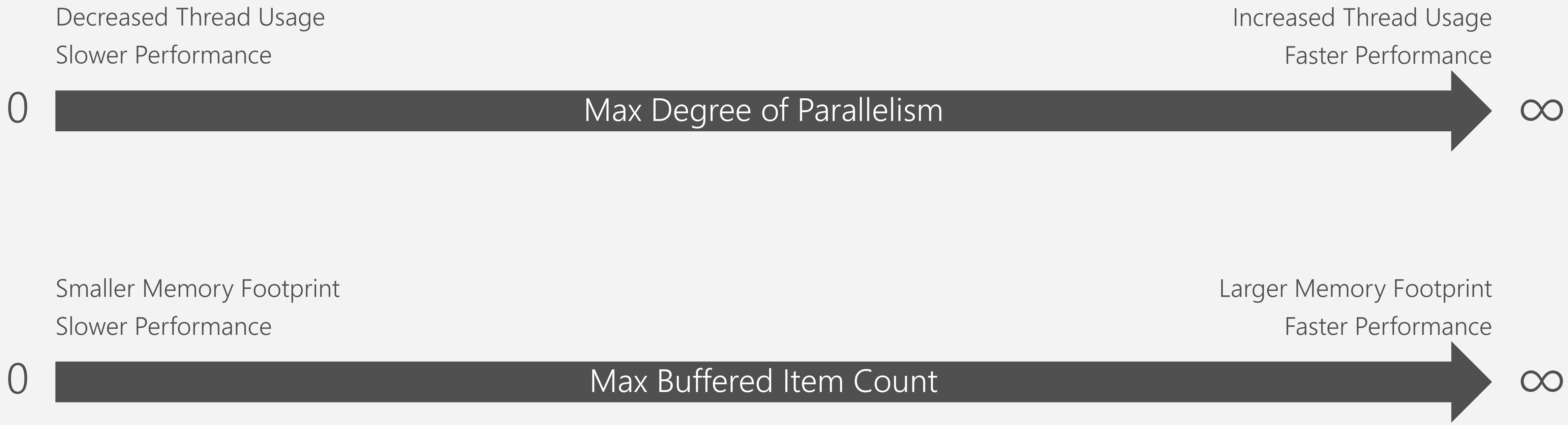


CLIENT RESPONSE BUFFER



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

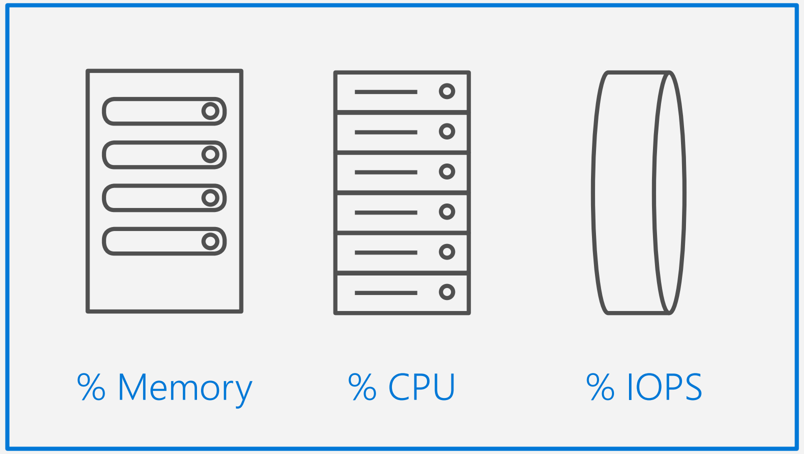
REQUEST UNITS

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



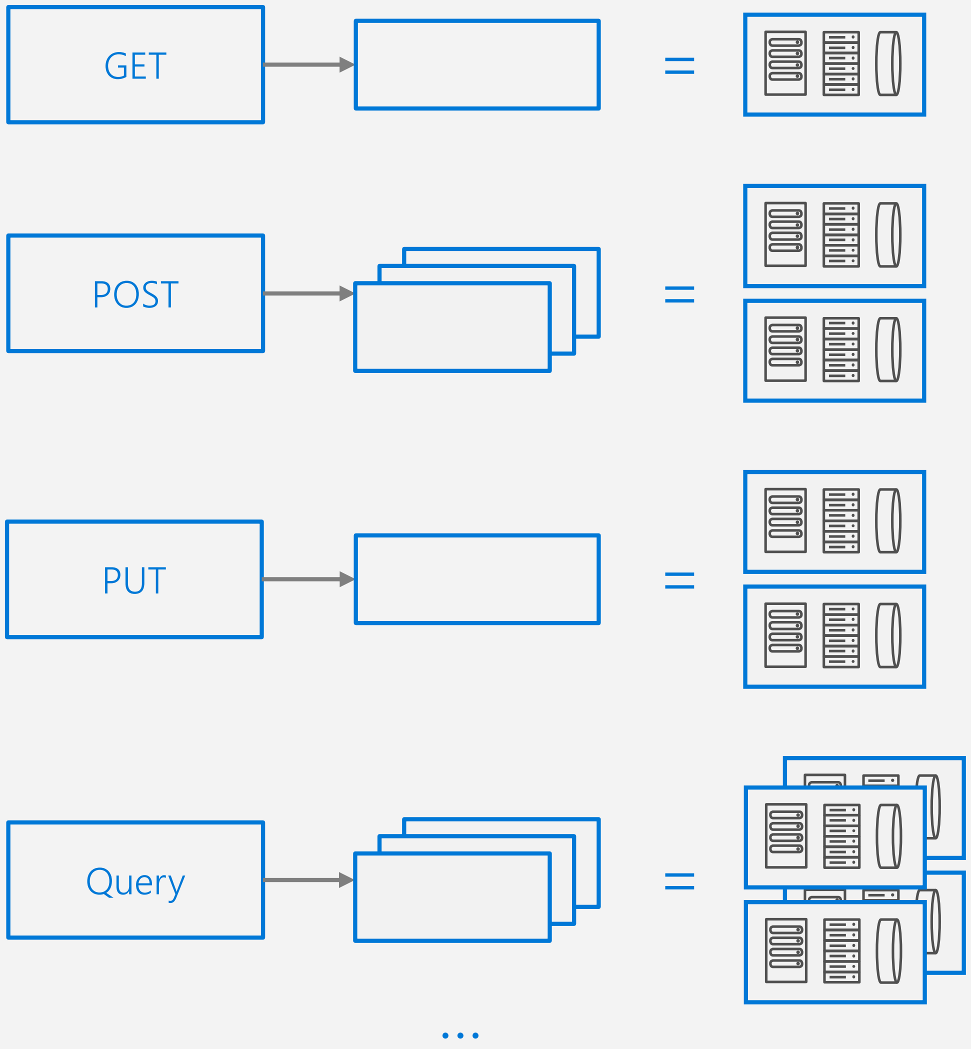
REQUEST UNITS

Normalized across various access methods

1 RU = 1 read of 1 KB document

Each request consumes fixed RUs

Applies to reads, writes, query, and stored procedures



REQUEST UNITS

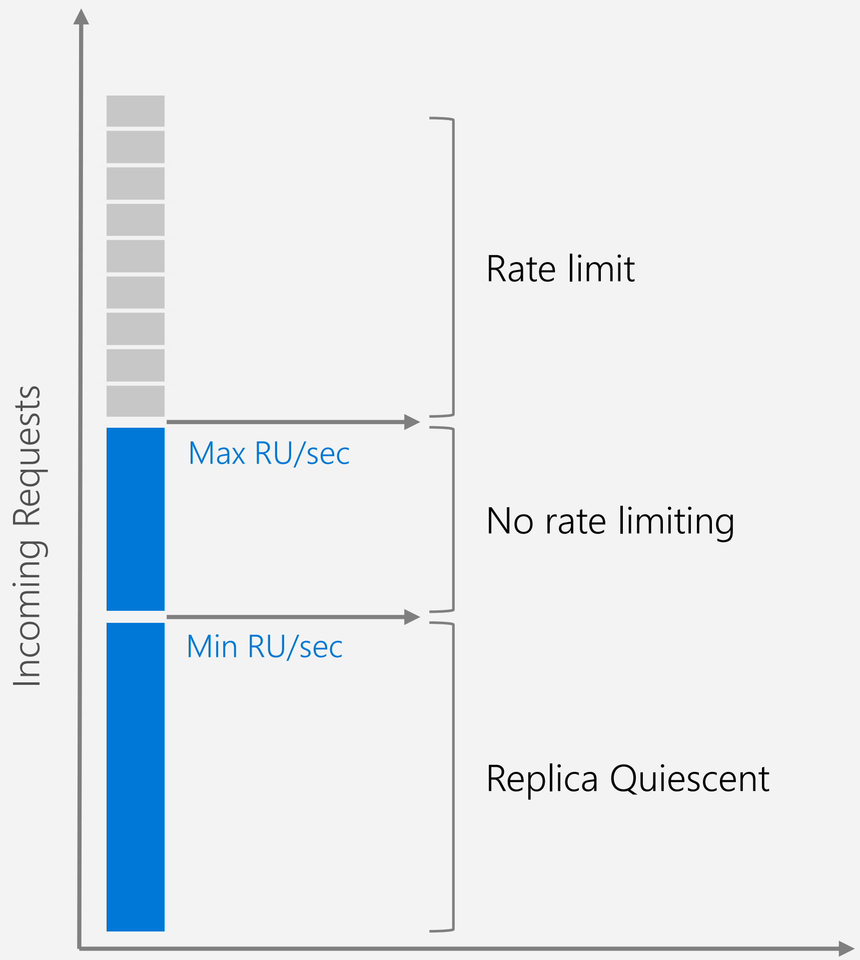
Provisioned in terms of RU/sec

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



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.

RU CHARGE MEASUREMENT EXAMPLE

```
ResourceResponse<Document> response = await client.CreateDocumentAsync(
        collectionLink,
        document
);
var requestUnits = response.RequestCharge;
```

.NET

RU CHARGE MEASUREMENT EXAMPLE

```
client.createDocument(
        collectionLink,
        documentDefinition,
        function (err, document, headers) {
                if (err) {
                        console.log(err);
                }
                var requestData = headers['x-ms-request-charge'];
        }
);
```

RU CHARGE MEASUREMENT EXAMPLE

```
ResourceResponse<Document> response = client.createDocument(
    collectionLink,
    documentDefinition,
    null,
    false
);

Double requestCharge = response.getRequestCharge();
```

REQUEST UNIT PRICING EXAMPLE

Storage Cost

Avg Record Size (KB)	1
Number of Records	100,000
Total Storage (GB)	100
Monthly Cost per GB	$0.25
Expected Monthly Cost for Storage	$25.00

Throughput Cost

Operation Type	Number of Requests per Second	Avg RU's per Request	RU's Needed
Create	100	5	500
Read	400	1	400

Total RU/sec	900
Monthly Cost per 100 RU/sec	$6.00
Expected Monthly Cost for Throughput	$54.00

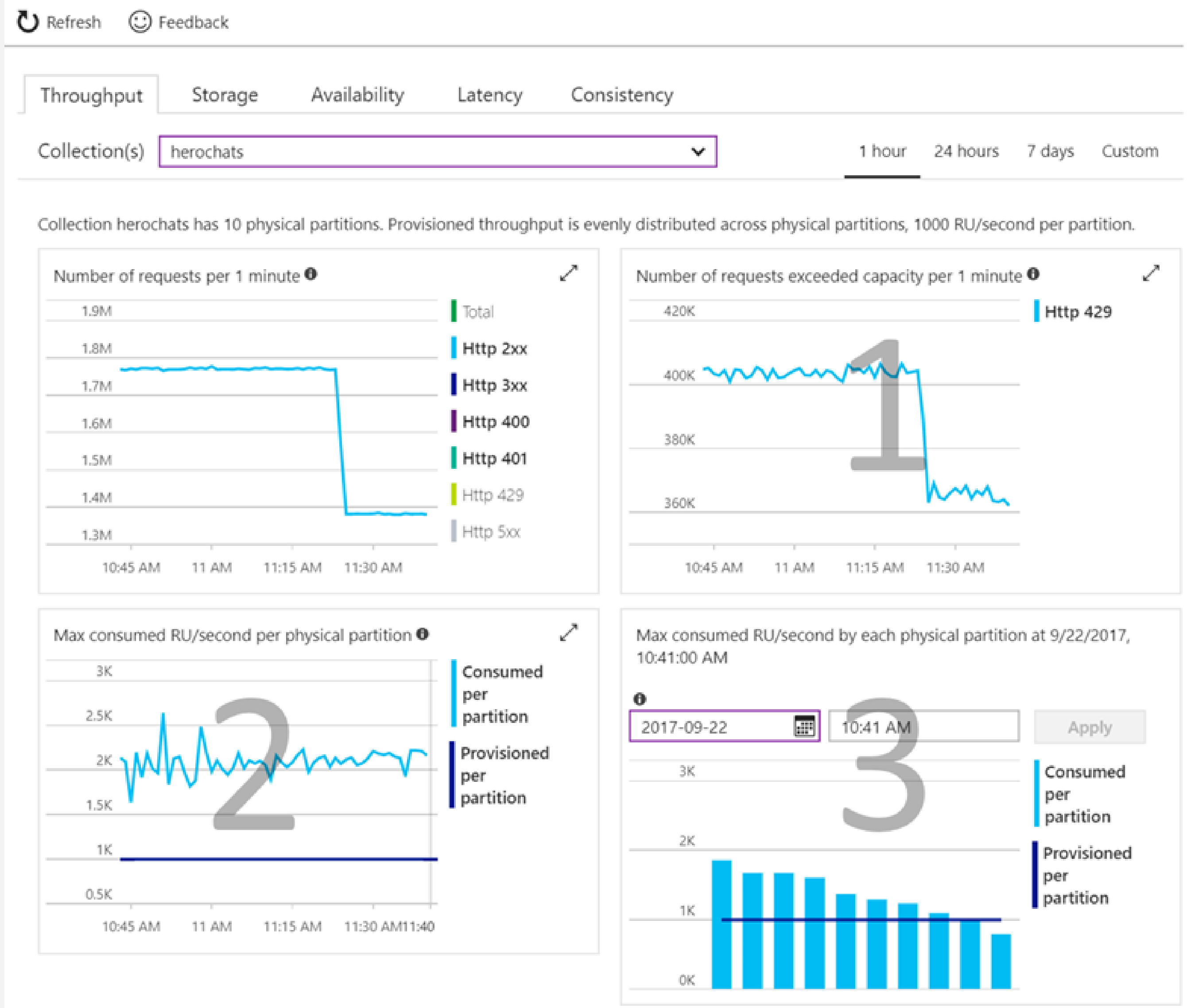
Total Monthly Cost

[Total Monthly Cost] = [Monthly Cost for Storage] + [Monthly Cost for Throughput]
= $25 + $54
= $79 per month

* pricing may vary by region; for up-to-date pricing, see: https://azure.microsoft.com/pricing/details/cosmos-db/

VALIDATING THROUGHPUT LEVEL CHOICE

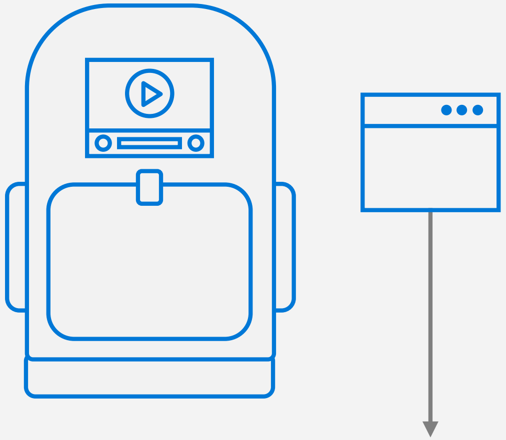
1. Check if your operations are getting rate limited.
   - Requests exceeding capacity chart
2. Check if consumed throughput exceeds the provisioned throughput on any of the physical partitions
   - Max RU/second consumed per partition chart
3. Select the time where the maximum consumed throughput per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



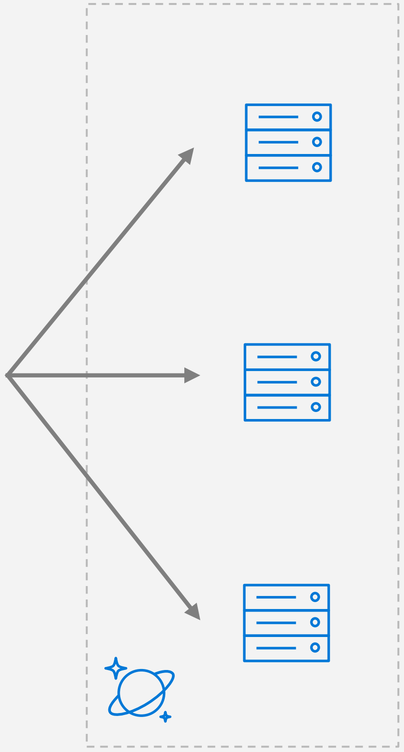
CROSS-PARTITION AGGREGATE

LOW-LATENCY AGGREGATION WORKS ACROSS MULTIPLE PARTITIONS

You can submit a simple SQL query and Azure Cosmos DB handles the routing of the query among data partitions and merges results to return the final aggregate values.



```
SELECT VALUE
    COUNT(1)
FROM
    telemetry t
WHERE
    t.deviceId = "craft267_seat17"
```



Example – Contoso Connected Car

BOUNDED LOCATION SEARCH USING GEO-DATA

GEOJSON SPECIFICATION

Azure Cosmos DB supports indexing and querying of geospatial point data that's represented using the GeoJSON specification.

SEARCH BY DISTANCE FROM POINT

The **ST_DISTANCE** built-in function returns the distance between the two GeoJSON Point expressions.

SEARCH WITHOUT BOUNDED POLYGON

The **ST_WITHIN** built-in function returns a Boolean indicating whether the first GeoJSON Point expression is within a GeoJSON Polygon expression.

```
{
        "type":"Point",
        "coordinates":[ 31.9, -4.8 ]
}
```

```
{
        "type":"Polygon",
        "coordinates":[[
                [ 31.8, -5 ],
                [ 31.8, -4.7 ],
                [ 32, -4.7 ],
                [ 32, -5 ],
                [ 31.8, -5 ]
    ]]
}
```

DISTANCE FROM CENTER POINT SEARCH

ST_DISTANCE

ST_DISTANCE can be used to measure the distance between two points. Commonly this function is used to determine if a point is within a specified range (meters) of another point.

```
SELECT *
FROM flights f
WHERE ST_DISTANCE(f.origin.location, {
        "type": "Point",
        "coordinates": [-122.19, 47.36]
}) < 100 * 1000
```

POLYGON SHAPE SEARCH

ST_WITHIN

ST_WITHIN can be used to check if a point lies within a Polygon. Commonly Polygons are used to represent boundaries like zip codes, state boundaries, or natural formations.

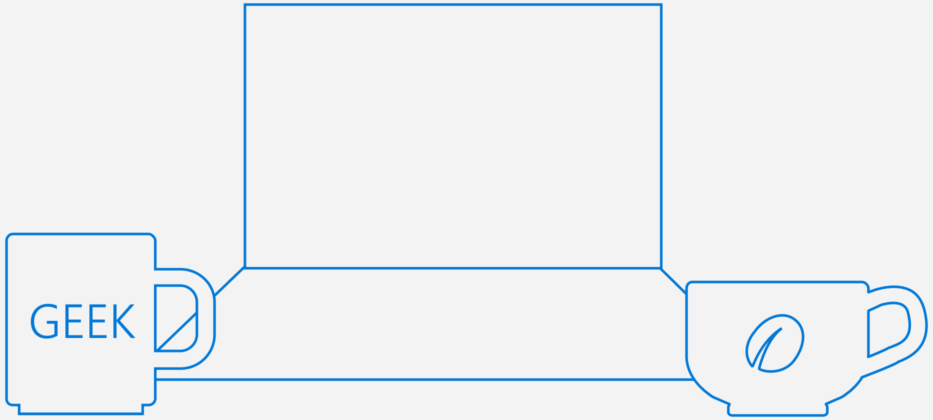
Polygon arguments in ST_WITHIN can contain only a single ring, that is, the Polygons must not contain holes in them.

```
SELECT *
FROM flights f
WHERE ST_WITHIN(f.destination.location,
{
        "type": "Polygon",
        "coordinates": [[
                [-124.63, 48.36],
                [-123.87, 46.14],
                [-122.23, 45.54],
                [-119.17, 45.95],
                [-116.92, 45.96],
                [-116.99, 49.00],
                [-123.05, 49.02],
                [-123.15, 48.31],
                [-124.63, 48.36]
        ]]
})
```

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.

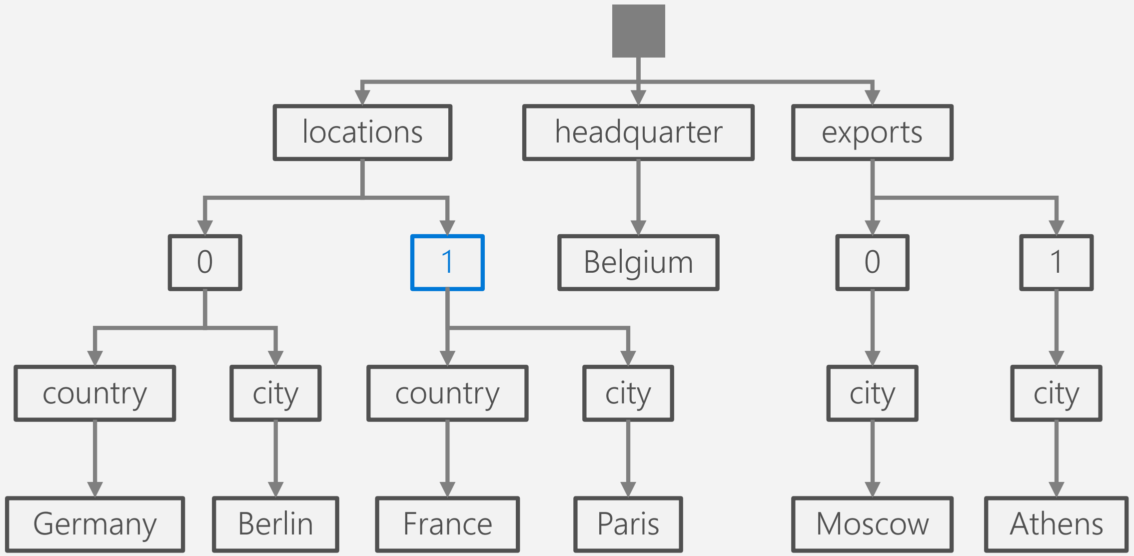
- Automatic index management
- Synchronous auto-indexing
- No schemas or secondary indices needed
- Works across every data model



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

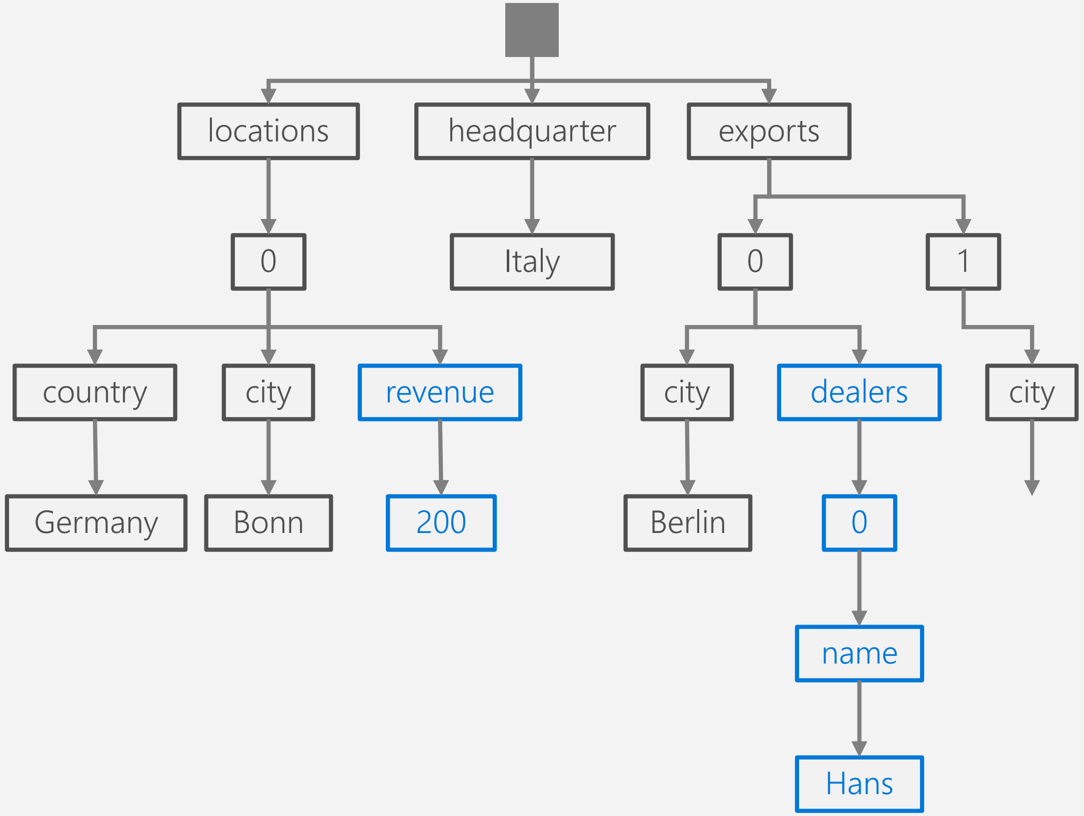
INDEXING JSON DOCUMENTS

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

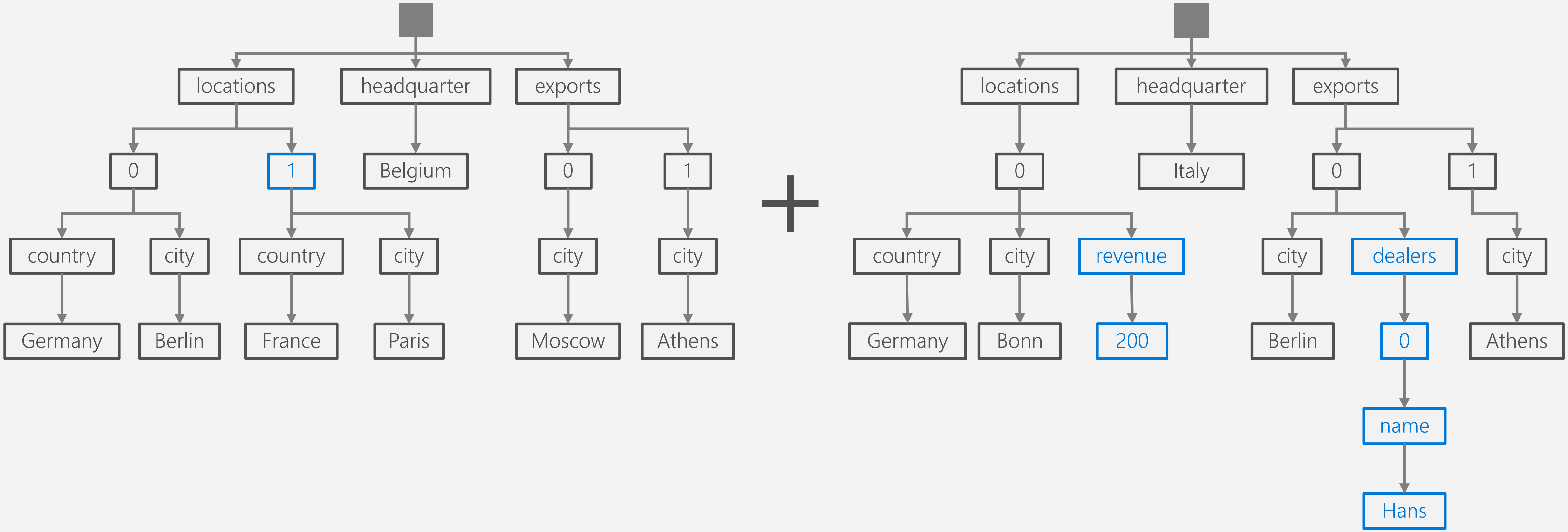


INDEXING JSON DOCUMENTS

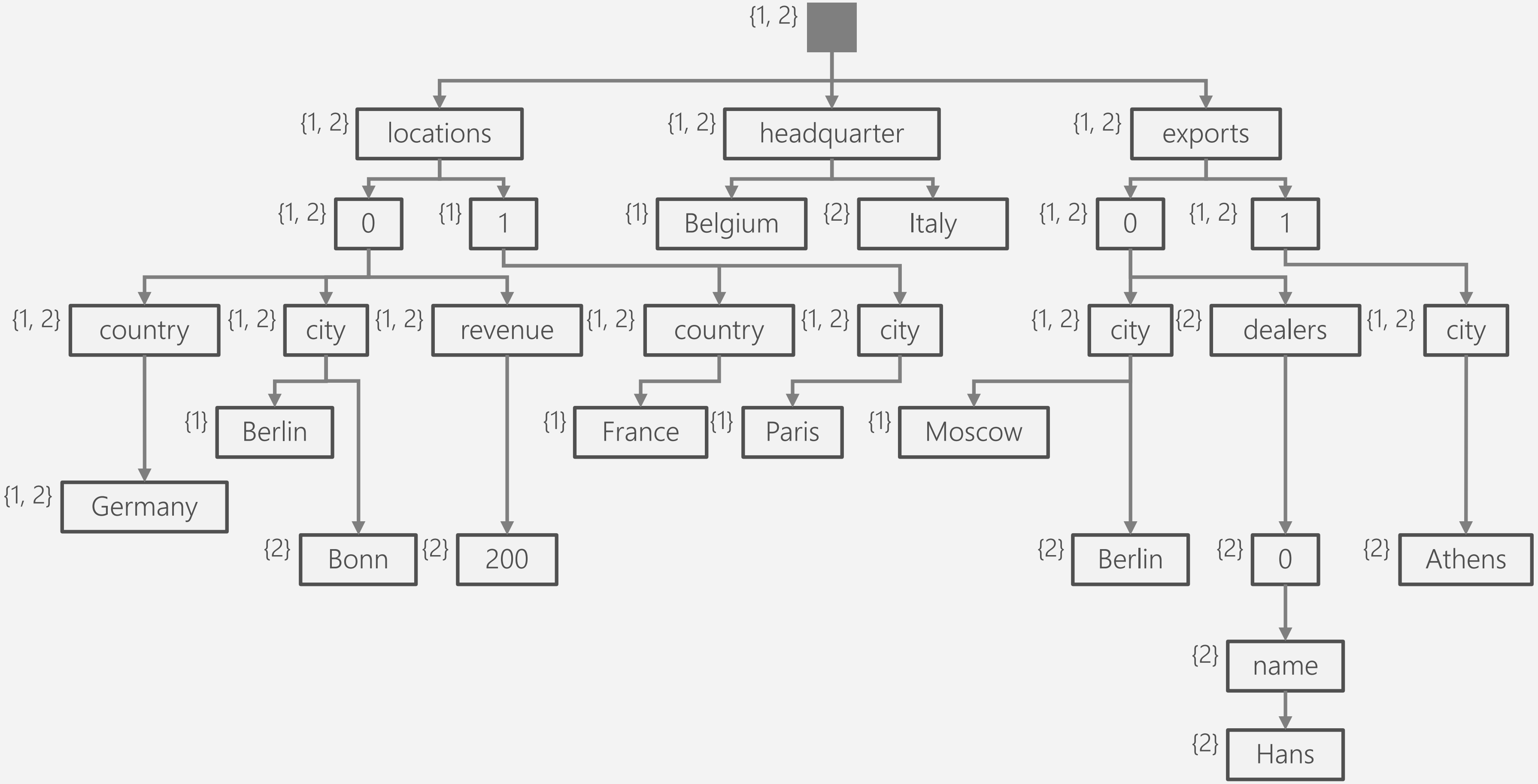
```
{
    "locations": [
        {
            "country": "Germany",
            "city": "Bonn",
            "revenue": 200
        }
    ],
    "headquarter": "Italy",
    "exports": [
        {
            "city": "Berlin",
            "dealers": [
                { "name": "Hans" }
            ]
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

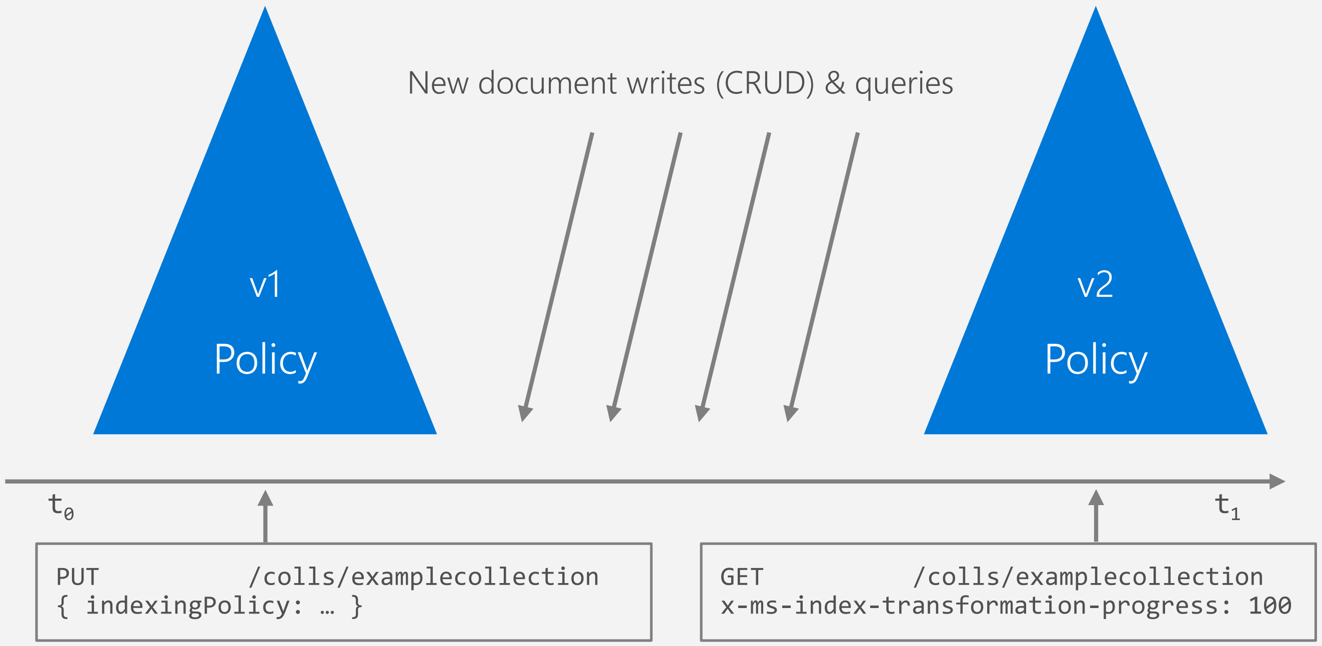
```
{
    "automatic": true,
    "indexingMode": "Consistent",
    "includedPaths": [{
        "path": "/*",
        "indexes": [{
            "kind": "Hash",
            "dataType": "String",
            "precision": -1
        }, {
            "kind": "Range",
            "dataType": "Number",
   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

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.



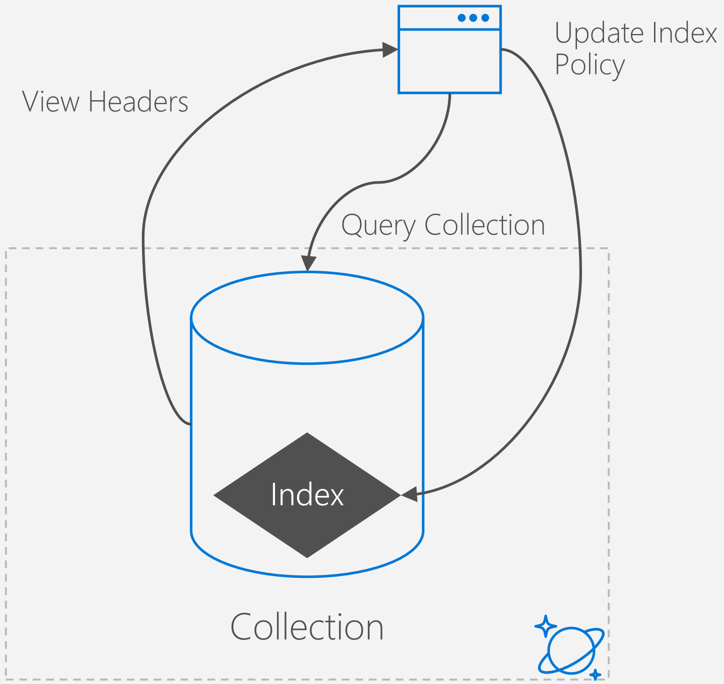
INDEX TUNING

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.

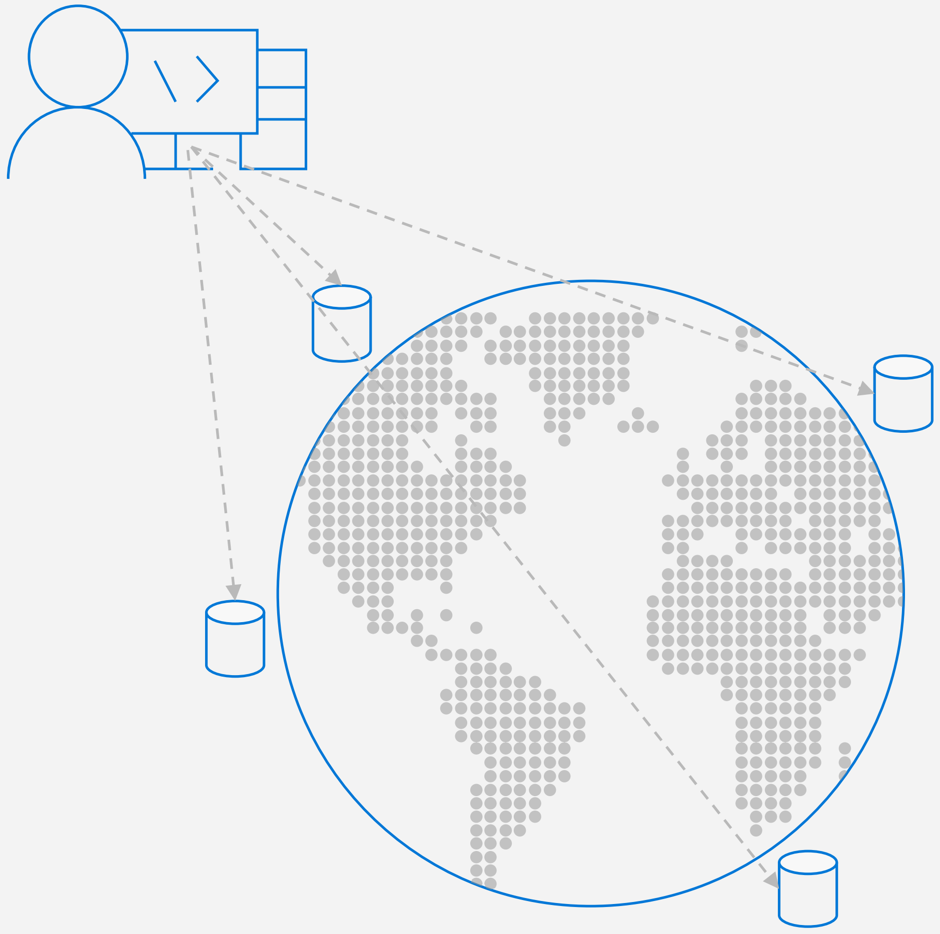


SQL Syntax

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.



SQL QUERY SYNTAX

BASIC QUERY SYNTAX

The **SELECT** & **FROM** keywords are the basic components of every query.

```
SELECT
    tickets.id,
    tickets.pricePaid
FROM tickets
```

```
SELECT
    t.id,
    t.pricePaid
FROM tickets t
```

SQL QUERY SYNTAX - WHERE

FILTERING

WHERE supports complex scalar expressions including arithmetic, comparison and logical operators

```
SELECT
    tickets.id,
    tickets.pricePaid
FROM tickets
WHERE
    tickets.pricePaid > 500.00 AND
    tickets.pricePaid <= 1000.00
```

SQL QUERY SYNTAX - PROJECTION

JSON PROJECTION

If your workloads require a specific JSON schema, Azure Cosmos DB supports JSON projection within its queries

```
SELECT {
    "id": tickets.id,
    "flightNumber": tickets.assignedFlight.flightNumber,
    "purchase": {
        "cost": tickets.pricePaid
    },
    "stops": [
        tickets.assignedFlight.origin,
        tickets.assignedFlight.destination
    ]
} AS ticket
FROM tickets
```

```
[
    {
        "ticket": {
            "id": "6ebe1165836a",
            "purchase": {
                "cost": 575.5
            },
            "stops": [
                "SEA",
                "JFK"
            ]
        }
    }
]
```

SQL QUERY SYNTAX - PROJECTION

SELECT VALUE

The **VALUE** keyword can further flatten the result collection if needed for a specific application workload

```
SELECT VALUE {
    "id": tickets.id,
    "flightNumber": tickets.assignedFlight.flightNumber,
    "purchase": {
        "cost": tickets.pricePaid
    },
    "stops": [
        tickets.assignedFlight.origin,
        tickets.assignedFlight.destination
    ]
}
FROM tickets
```

```
[
    {
        "id": "6ebe1165836a",
        "purchase": {
            "cost": 575.5
        },
        "stops": [
            "SEA",
            "JFK"
        ]
    }
]
```

INTRA-DOCUMENT JOIN

Azure Cosmos DB supports intra-document JOIN's for de-normalized arrays

Let's assume that we have two JSON documents in a collection:

```
{
    "pricePaid": 575.5,
    "assignedFlight": {
        "number": "F125",
        "origin": "SEA",
        "destination": "JFK"
    },
    "seat": "12A",
    "requests": [
        "kosher_meal",
        "aisle_seat"
    ],
    "id": "6ebe1165836a"
}
```

We are interested in querying an array internal to the document

```
{
    "pricePaid": 234.75,
    "assignedFlight": {
        "number": "F752",
        "origin": "SEA",
        "destination": "LGA"
    },
    "seat": "14C",
    "requests": [
        "early_boarding",
        "window_seat"
    ],
    "id": "c4991b4d2efc"
}
```

SQL

INTRA-DOCUMENT JOIN

We can filter on a particular array index position without JOIN:

```sql
SELECT
    tickets.assignedFlight.number,
    tickets.seat,
    ticket.requests
FROM
    tickets
WHERE
    ticket.requests[1] == "aisle_seat"
```

```json
[
    {
        "number":"F125","seat":"12A",
        "requests": [
            "kosher_meal",
            "aisle_seat"
        ]
    }
]
```

SQL

INTRA-DOCUMENT JOIN

JOIN allows us to merge embedded documents or arrays across multiple documents and returned a flattened result set:

```sql
SELECT
    tickets.assignedFlight.number,
    tickets.seat,
    requests
FROM
    tickets
JOIN
    requests IN tickets.requests
```

```json
[
    {
        "number":"F125","seat":"12A",
        "requests":"kosher_meal"
    },
    {
        "number":"F125","seat":"12A",
        "requests":"aisle_seat"
    },
    {
        "number":"F752","seat":"14C",
        "requests":"early_boarding"
    },
    {
        "number":"F752","seat":"14C",
        "requests":"window_seat"
    }
]
```

SQL

INTRA-DOCUMENT JOIN

Along with JOIN, we can also filter the cross products without knowing the array index position:

```
SELECT
    tickets.id, requests
FROM
    tickets
JOIN
    requests IN tickets.requests
WHERE
    requests
    IN ("aisle_seat", "window_seat")
```

```
[
    {
        "number":"F125","seat":"12A“,
        "requests": "aisle_seat"
    },
    {
        "number":"F752","seat":"14C",
        "requests": "window_seat"
    }
]
```

SQL

PAGINATED QUERY RESULTS

Straightforward approach to paginate the results:

```
var query = client.CreateDocumentQuery<ExampleEntity>(collectionUri, options);
var docQuery = query.AsDocumentQuery();

List<ExampleEntity> results = new List<ExampleEntity>();
while (docQuery.HasMoreResults)
{
    foreach (ExampleEntity item in await query.ExecuteNextAsync())
    {
        results.Add(item);
    }
}
```

.NET

PAGINATED QUERY RESULTS

Pagination with ToList():

```
var query = client.CreateDocumentQuery<ExampleEntity>(collectionUri, options);
var docQuery = query.AsDocumentQuery();

List<ExampleEntity> results = new List<ExampleEntity>();
results = query.ToList();
```

ToList() automatically iterates through all pages

.NET

PAGINATED QUERY RESULTS

Pagination with hasNext() in Java:

```
Iterator<Document> documents = client.queryDocuments(
    collectionLink,
    queryString,
    options
).getQueryIterator();

while(documents.hasNext()) {
    Document current = documents.next();
}
```

SQL QUERY PARAMETRIZATION

```
var query = client.CreateDocumentQuery<ExampleEntity>(collectionUri,
    new SqlQuerySpec
    {
        QueryText = "SELECT * FROM dataset s WHERE (s.id = @id)",
        Parameters = new SqlParameterCollection
        {
            new SqlParameter("@id", "exampleIdentifier")
        }
    }, options
);

List<ExampleEntity> results = new List<ExampleEntity>();
results = query.ToList<ExampleEntity>();
```

.NET

SQL QUERY IN LINQ

```
var query = client.CreateDocumentQuery<ExampleEntity>(collectionUri, options);

var docQuery = query
                .Where(s => s.LastName == "Andersen")
                .Select(s => new { Name = s.LastName })
                .AsDocumentQuery();

List<ExampleEntity> results = new List<ExampleEntity>();
results = query.ToList<ExampleEntity>();
```

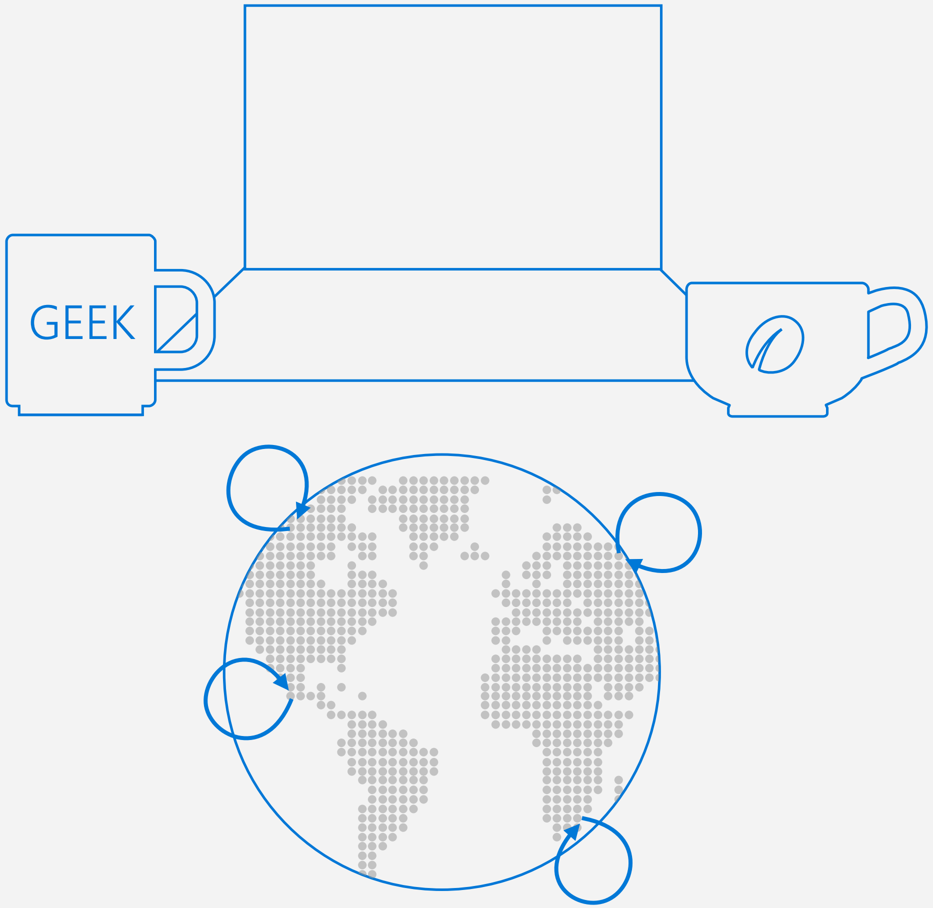
.NET

Programming

PROGRAMMING

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

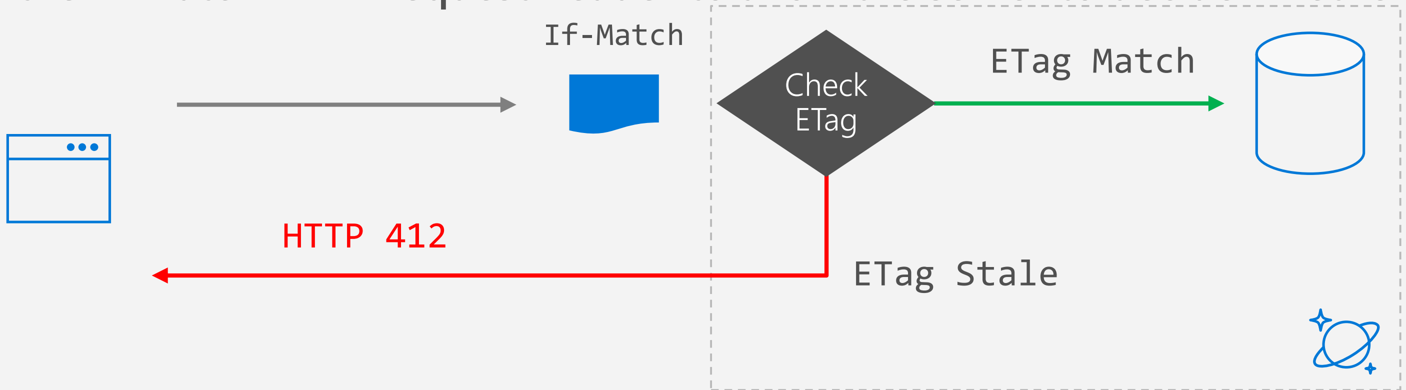
This module will reference programming in the context of the SQL API.



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:**



CONTROL CONCURRENCY USING ETAGS

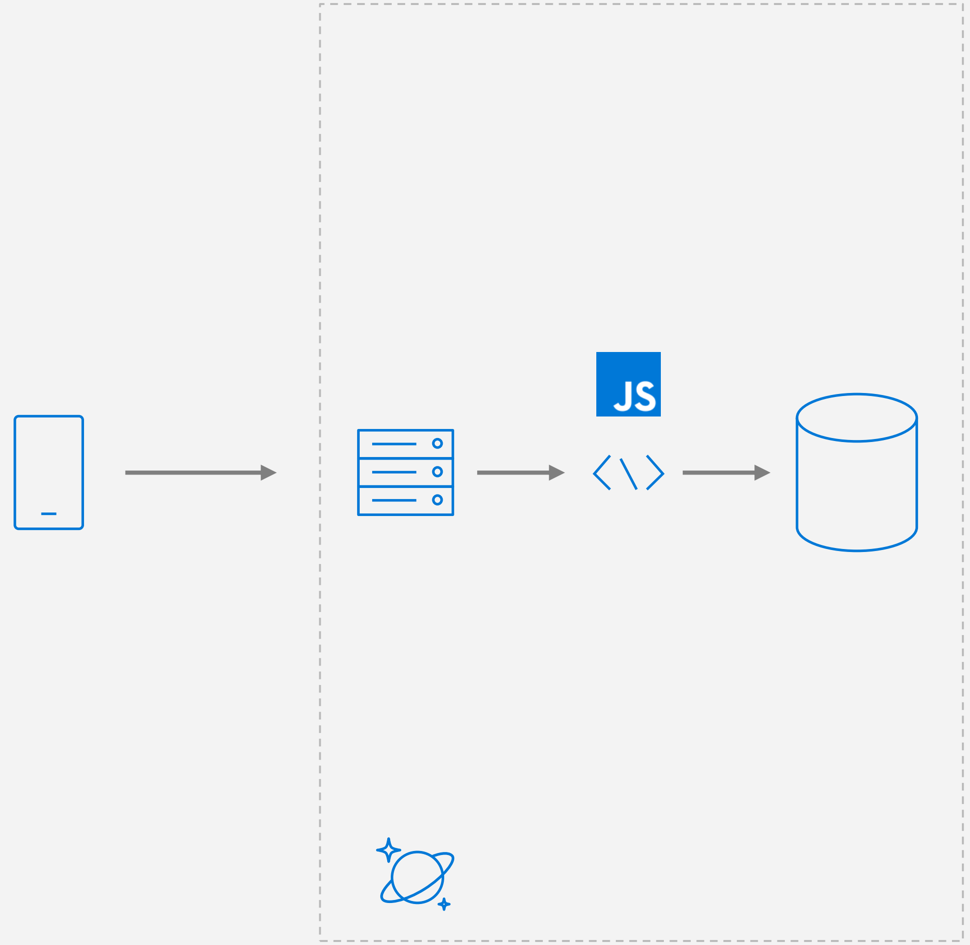
```
try
{
    var ac = new AccessCondition { Condition = readDoc.ETag, Type = AccessConditionType.IfMatch };
    updatedDoc = await client.ReplaceDocumentAsync(readDoc, new RequestOptions { AccessCondition = ac });
}
catch (DocumentClientException dce)
{
    if (dce.StatusCode == HttpStatusCode.PreconditionFailed)
    {
        Console.WriteLine("Another concurrent process has updated the record");
    }
}
```

.NET

STORED PROCEDURES

BENEFITS

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



SIMPLE STORED PROCEDURE

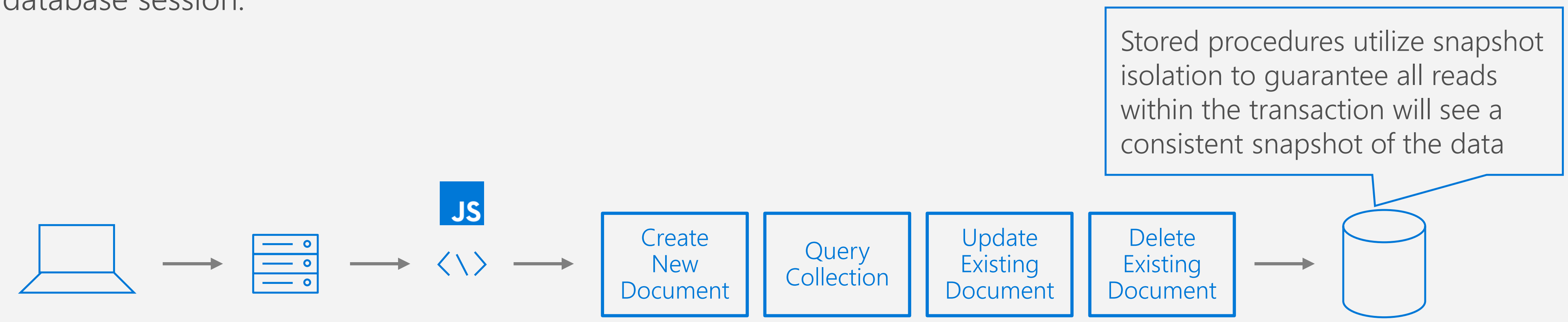
```
function createSampleDocument(documentToCreate) {
	var context = getContext();
	var collection = context.getCollection();
	var accepted = collection.createDocument(
		collection.getSelfLink(),
		documentToCreate,
		function (error, documentCreated) {
			context.getResponse().setBody(documentCreated.id)
		}
	);
	if (!accepted) return;
}
```

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.



BOUNDED EXECUTION

EXECUTION WITHIN TIME BOUNDARIES

All Azure Cosmos DB operations must complete within the server-specified request timeout duration. If an operation does not complete within that time limit, the transaction is rolled back.

HELPER BOOLEAN VALUE

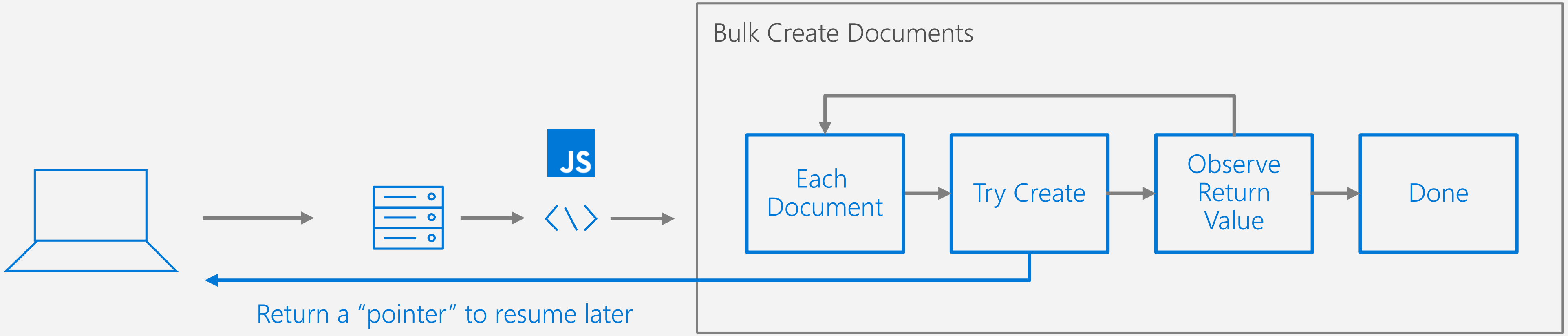
All functions under the collection object (for create, read, replace, and delete of documents and attachments) return a **Boolean value** that represents whether that operation will complete:

- **If true**, the operation is expected to complete
- **If false**, the time limit will soon be reached and your function should end execution as soon as possible.

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"**



CONTROL FLOW

JAVASCRIPT CONTROL FLOW

Stored procedures allow you to naturally express control flow, variable scoping, assignment, and integration of exception handling primitives with database transactions directly in terms of the JavaScript programming language.

ES6 PROMISES

ES6 promises can be used to implement promises for Azure Cosmos DB stored procedures. Unfortunately, **promises “swallow” exceptions by default**. It is recommended to use callbacks instead of ES6 promises.

STORED PROCEDURE CONTROL FLOW

```
function createTwoDocuments(docA, docB) {
    var ctxt = getContext(); var coll = context.getCollection(); var collLink = coll.getSelfLink();
    var aAccepted = coll.createDocument(collLink, docA, docACallback);
    function docACallback(error, created) {
        var bAccepted = coll.createDocument(collLink, docB, docBCallback);
        if (!bAccepted) return;
    };
    function docBCallback(error, created) {
        context.getResponse().setBody({
            "firstDocId": created.id,
            "secondDocId": created.id
        });
    };
}
```

STORED PROCEDURE CONTROL FLOW

```
function createTwoDocuments(docA, docB) {
    var ctxt = getContext(); var coll = context.getCollection(); var collLink = coll.getSelfLink();
    var aAccepted = coll.createDocument(collLink, docA, function(docAError, docACreated) {
        var bAccepted = coll.createDocument(collLink, docB, function(docBError, docBCreated) {
            context.getResponse().setBody({
                "firstDocId": docACreated.id,
                "secondDocId": docBCreated.id
            });
        });
        if (!aAccepted) return;
    });
    if (!bAccepted) return;
}
```

Nesting your callbacks is just as valid of a method as using named callback functions

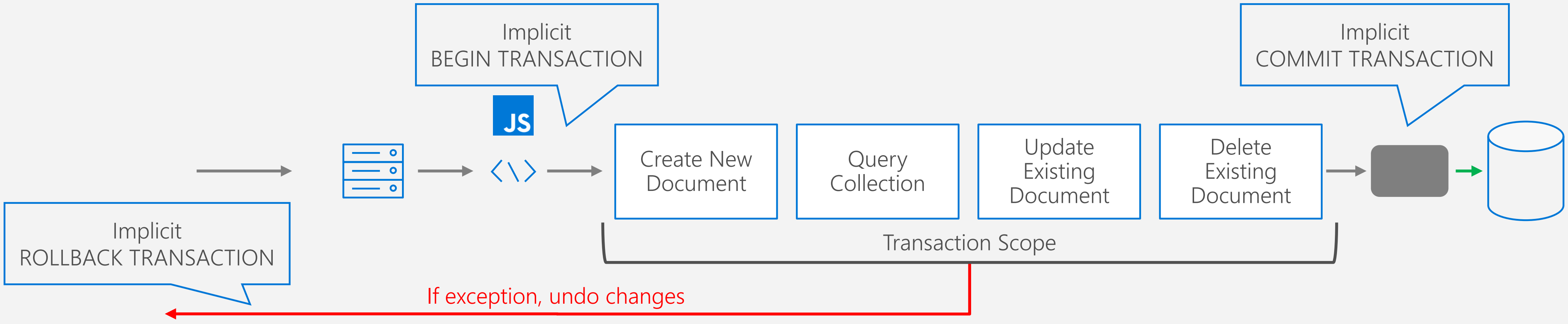
JS

ROLLING BACK TRANSACTIONS

TRANSACTION ROLL-BACK

Inside a JavaScript function, all operations are automatically wrapped under a single transaction:

- If the function completes without any exception, all data changes are committed
- If there is any exception that's thrown from the script, Azure Cosmos DB's JavaScript runtime will roll back the whole transaction.



TRANSACTION ROLLBACK IN STORED PROCEDURE

```
collection.createDocument(
        collection.getSelfLink(),
        documentToCreate,
        function (error, documentCreated) {
                if (error) throw "Unable to create document, aborting...";
        }
);
collection.createDocument(
        documentToReplace._self,
        replacementDocument,
        function (error, documentReplaced) {
                if (error) throw "Unable to update document, aborting...";
        }
);
```

DEBUGGING STORED PROCEDURES

CONSOLE LOGGING

Much like with traditional JavaScript applications, you can use **console.log()** to capture various telemetry and data points for your running code.

VIEWING SCRIPT LOGS

.NET

You must **opt-in** to viewing and capturing console output using the **EnableScriptLogging** boolean property available in the client SDK. The SDK has a **ScriptLog** property on the **StoredProcedureResponse** class that contains the captured output of the JavaScript console log.

DEBUGGING STORED PROCEDURES

```
var response = await client.ExecuteStoredProcedureAsync(
    document.SelfLink,
    new RequestOptions
    {
        EnableScriptLogging = true
    }
);
String logData = response.ScriptLog;
```

.NET

DEBUGGING STORED PROCEDURES

```
RequestOptions requestOptions = new RequestOptions();
requestOptions.EnableScriptLogging = true;

StoredProcedureResponse response = client.executeStoredProcedure(
    storedProcLink,
    requestOptions,
    new Object[]{}
);
```

USER-DEFINED FUNCTIONS

UDF

- User-defined functions (UDFs) are used to extend the Azure Cosmos DB SQL API's query language grammar and implement custom business logic. UDFs can only be called from inside queries
- They do not have access to the context object and are meant to be used as compute-only code

USER-DEFINED FUNCTION DEFINITION

```
var taxUdf = {
        id: "tax",
        serverScript: function tax(income) {
                if (income == undefined)
                        throw 'no input’;
                if (income < 1000)
                        return income * 0.1;
                else if (income < 10000)
                        return income * 0.2;
                else
                        return income * 0.4;
        }
}
```

USER-DEFINED FUNCTION USAGE IN QUERIES

```sql
SELECT
      *
FROM
      TaxPayers t
WHERE
      udf.tax(t.income) > 20000
```

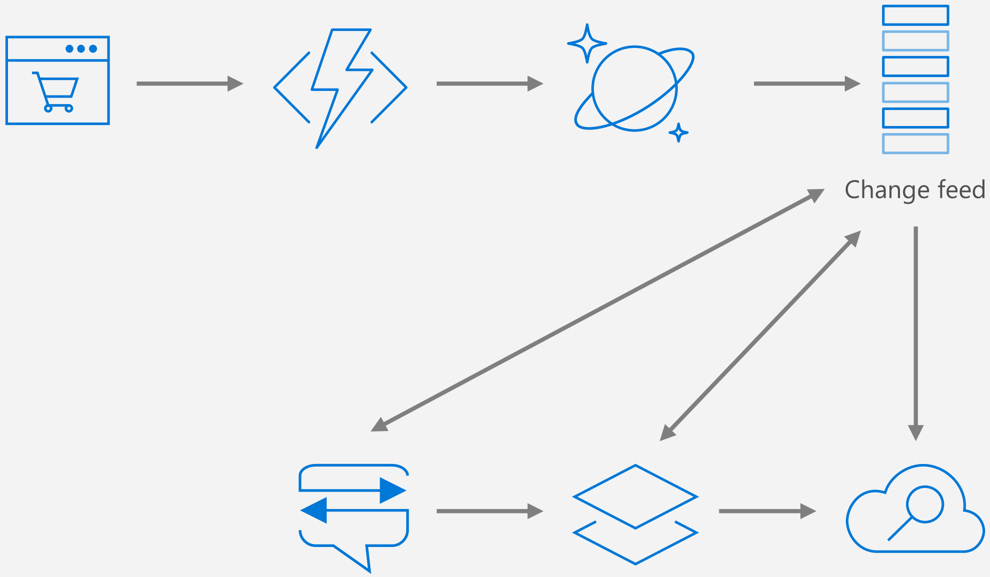
SQL

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.

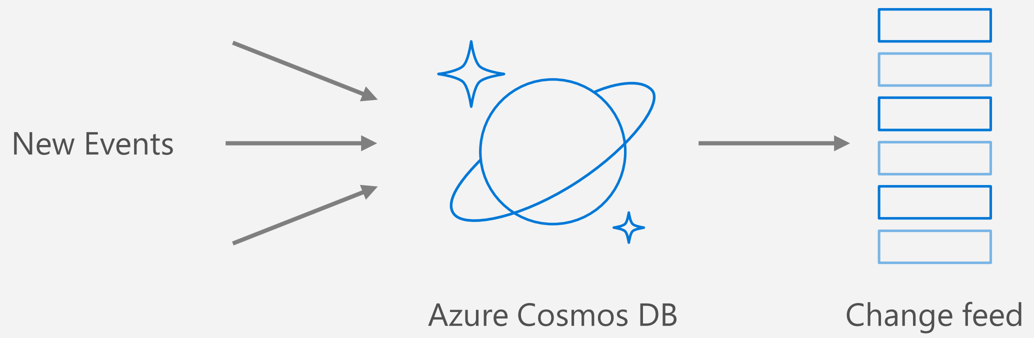
COMMON SCENARIOS

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

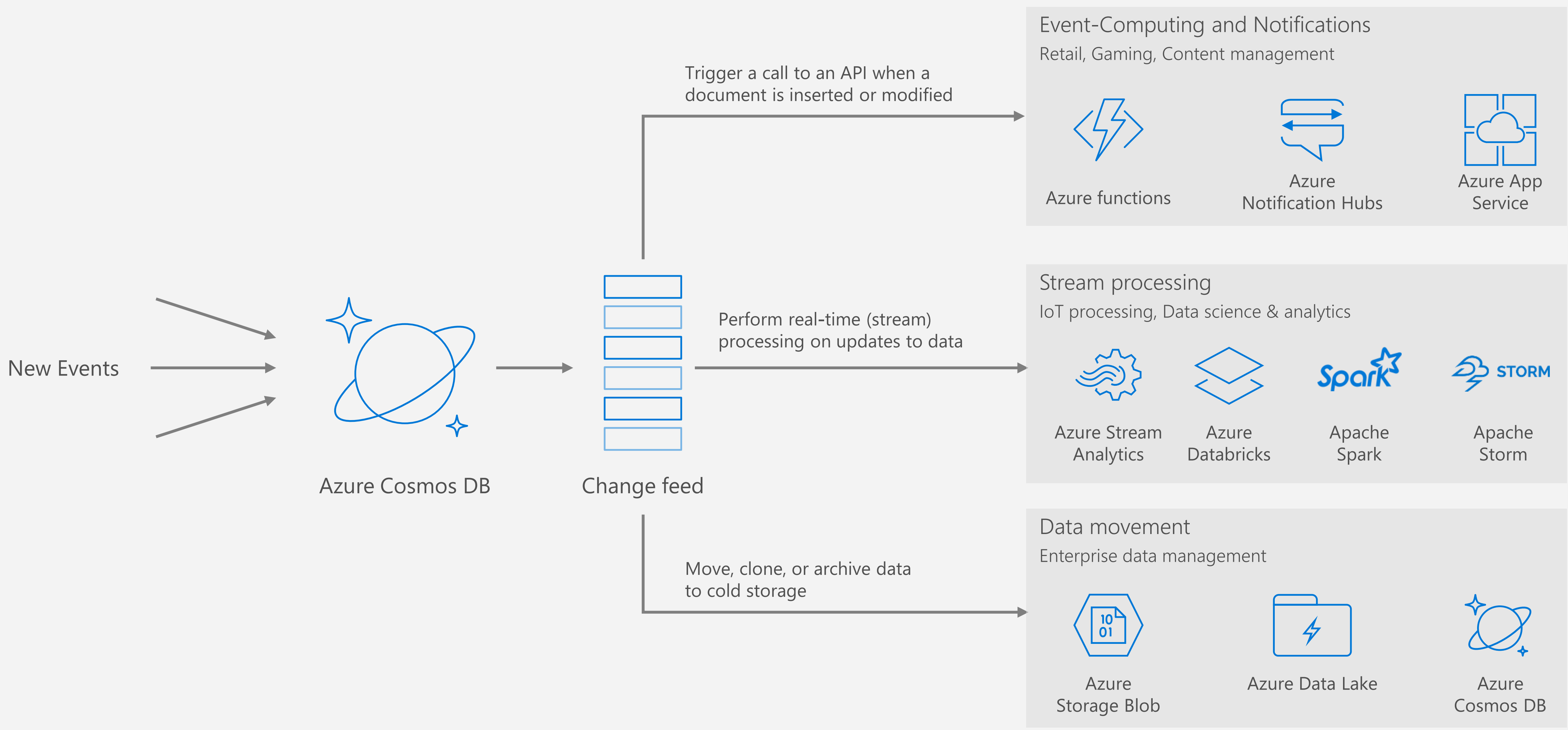


CHANGE FEED

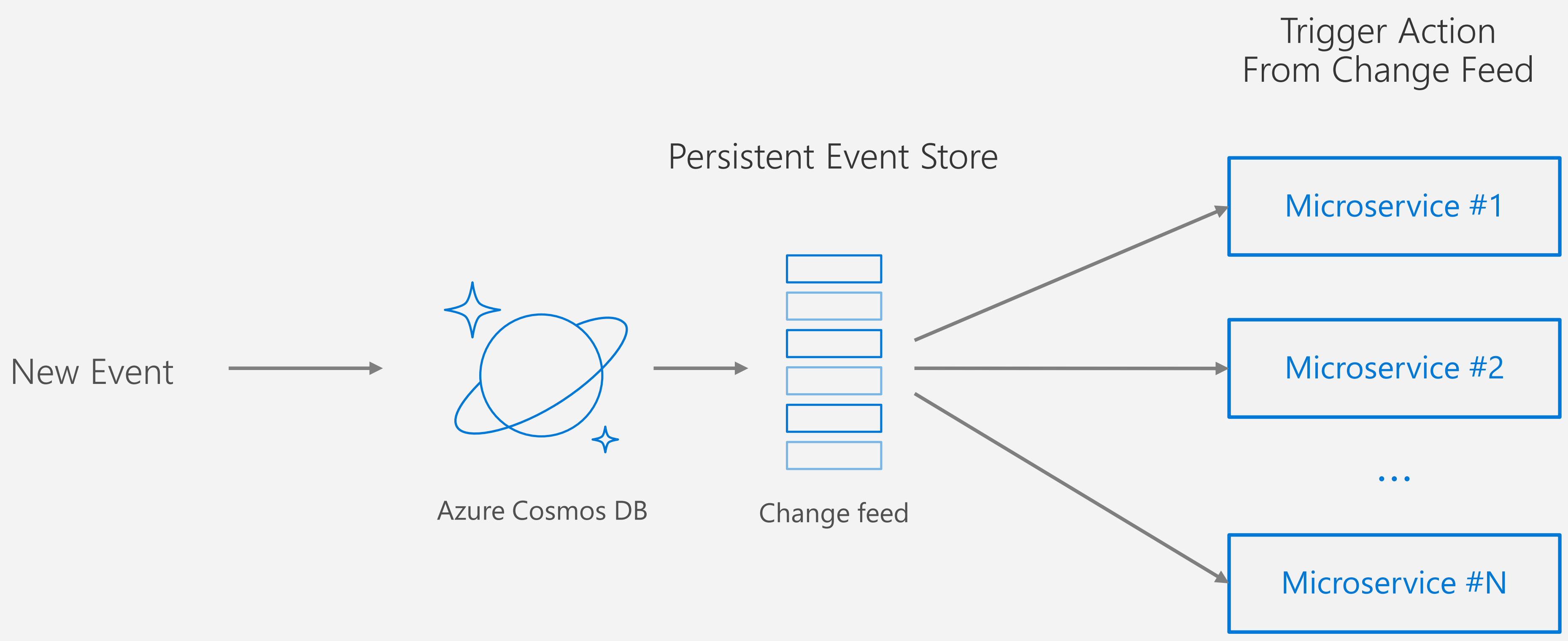
Persistent log of records within an Azure Cosmos DB container. Preseneted in the order in which they were modified



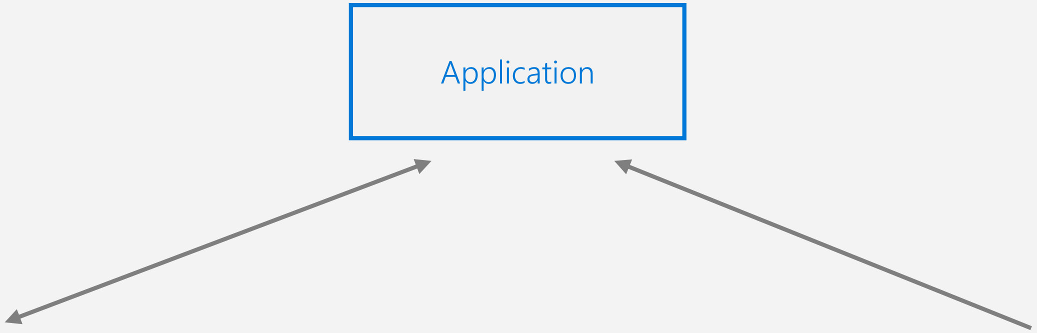
CHANGE FEED SCENARIOS



EVENT SOURCING FOR MICROSERVICES



MATERIALIZING VIEWS



Azure Cosmos DB

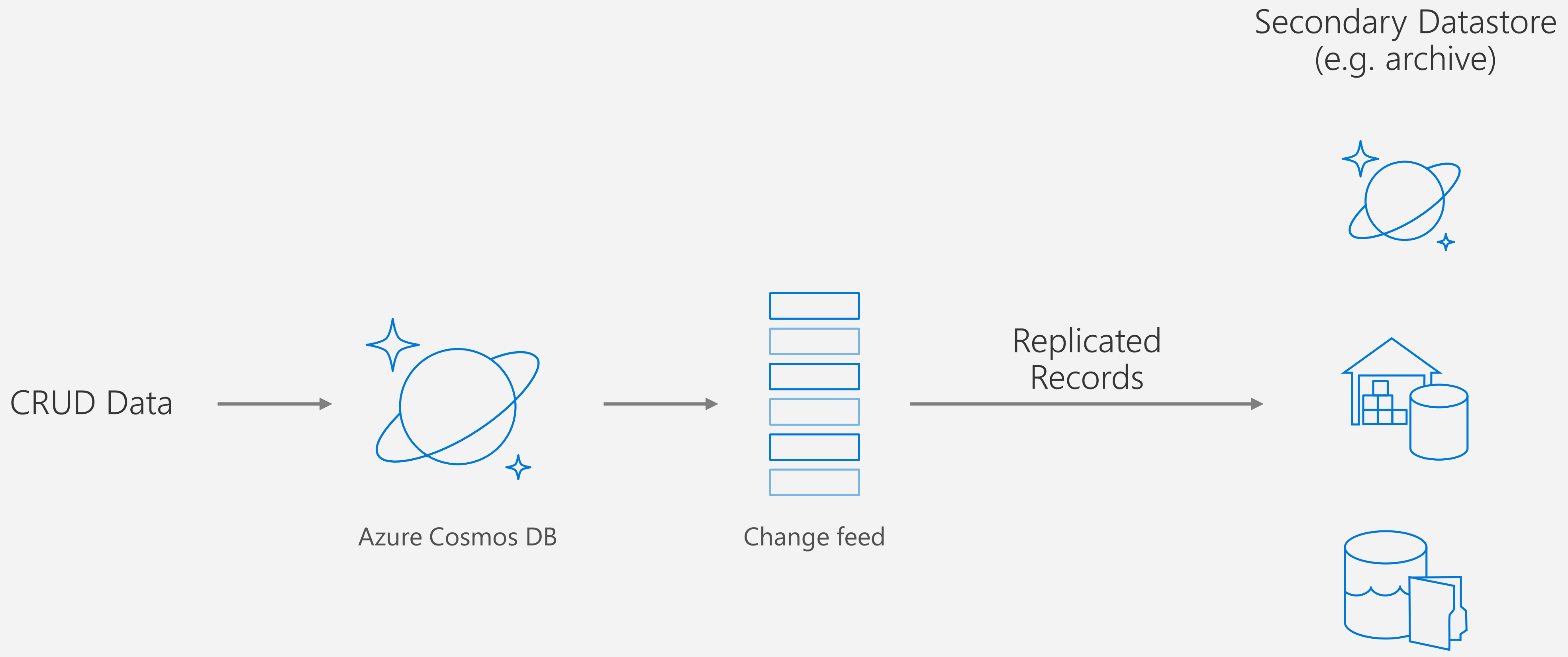
Subscription	User	Create Date	...
123abc	Ben6	6/17/17	
456efg	Ben6	3/14/17	
789hij	Jen4	8/1/16	
012klm	Joe3	3/4/17	

Change feed

Materialized View

User	Total Subscriptions
Ben6	2
Jen4	1
Joe3	1

REPLICATING DATA

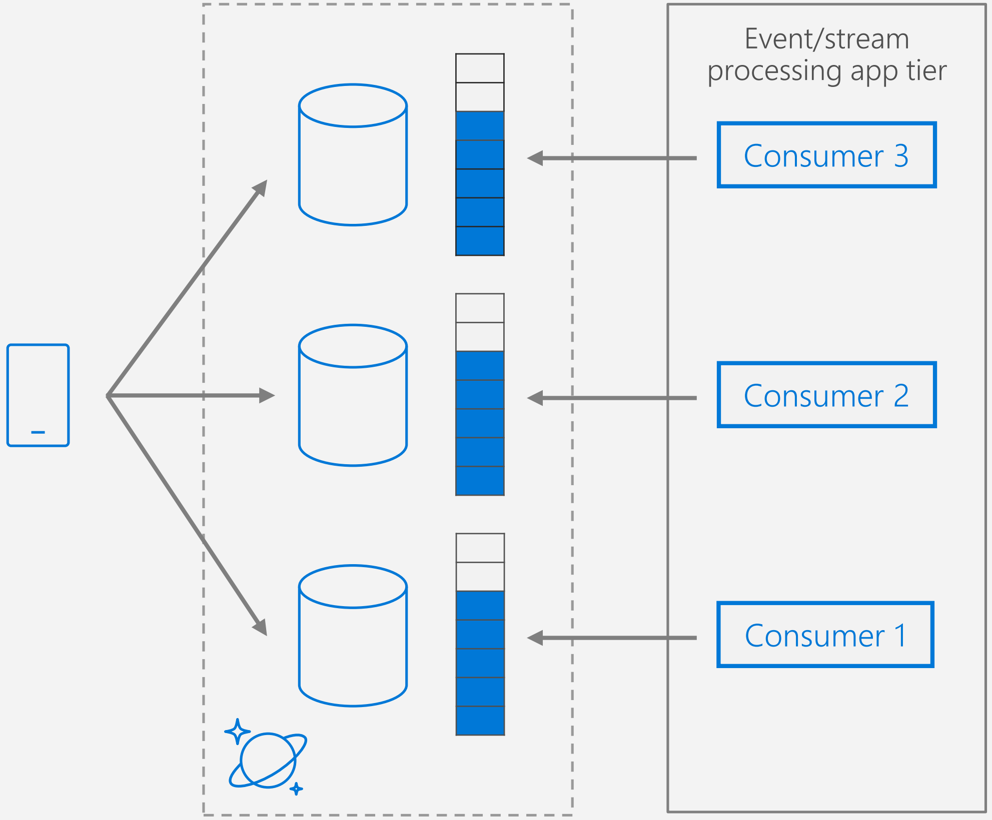


CHANGE FEED WITH PARTITIONS

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 – RETRIEVING KEY RANGES

```
string pkRangesResponseContinutation = null;
List<PartitionKeyRange> partitionKeyRanges = new List<PartitionKeyRange>();
do
{
        FeedResponse<PartitionKeyRange> pkPagesResponse = await client.ReadPartitionKeyRangeFeedAsync(
                collectionUri,
                new FeedOptions { RequestContinutation = pkRangesResponseContinutation }
        );
        partitionKeyRanges.AddRange(pkRangesResponse);
        pkRangesResponseContinutation = pkRangesResponse.ResponseContinutation;
}
while (pkRangesResponseContinutation != null);
```

.NET

CHANGE FEED - CONSUMING CHANGE FEED

```
foreach (PartitionKeyRange pkRange in partitionKeyRanges)
{
         string continuation = null;
         checkpoints.TryGetValue(pkRange.Id, out continutation);
         IDocumentQuery<Document> query = client.CreateDocumentChangeFeedQuery(
                  collection,
                  new ChangeFeedOptions
                  {
                           PartitionKeyRangeId = pkRange.Id,
                           StartFromBeginning = true,
                           RequestContinutation = continuation,
                           MaxItemCount = 1
                  }
         );
...
```

CONTINUED ON NEXT PAGE

.NET

CHANGE FEED PRIMITIVES - CONSUME CHANGE FEED

```
...
        while (query.HasMoreResults)
        {
                FeedResponse<DeviceReading> readChangesResponse = query.ExecuteNextAsync<DeviceReading>().Result;
                foreach (DeviceReading changedDocument in readChangesResponse)
                {
                        Console.WriteLine(changedDocument.Id);
                }
                checkpoints[pkRange.Id] = readChangesResponse.ResponseContinuation;
        }
}
```

.NET

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

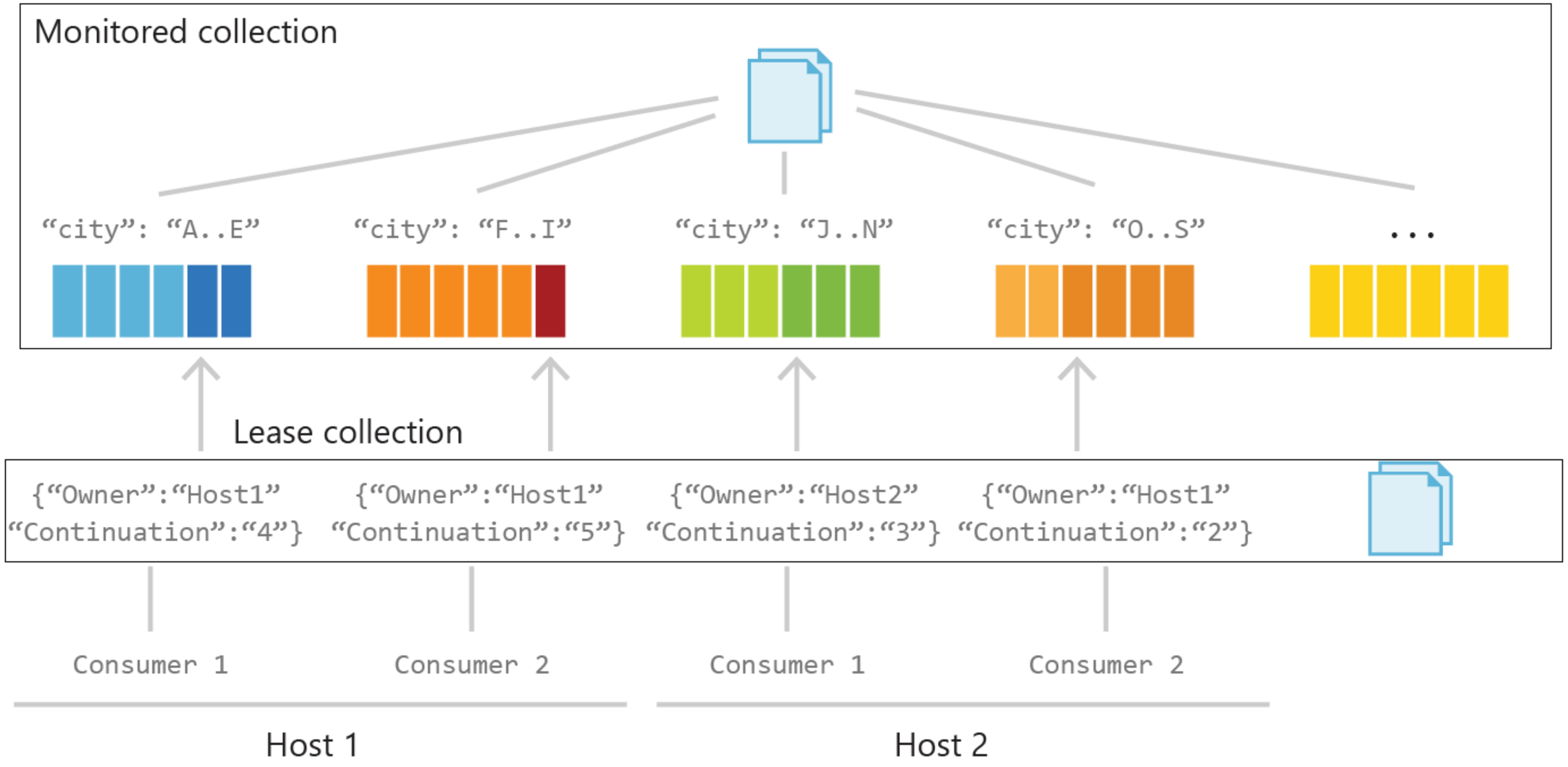
.NETFramework 4.5.2

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)

CHANGE FEED PROCESSOR – BEHIND THE SCENES



CHANGE FEED PROCESSOR – INTERFACE IMPLEMENTATION

```
public class DocumentFeedObserver : IChangeFeedObserver
{
    ...
    public Task IChangeFeedObserver.ProcessChangesAsync(ChangeFeedObserverContext context, IReadOnlyList<Document>
    docs)
    {
          Console.WriteLine("Change feed: {0} documents", Interlocked.Add(ref totalDocs, docs.Count));
          foreach(Document doc in docs)
          {
                   Console.WriteLine(doc.Id.ToString());
          }
          return Task.CompletedTask;
    }
}
```

.NET

CHANGE FEED PROCESSOR - REGISTRATION

```
DocumentFeedObserver docObserver = new DocumentFeedObserver();
ChangeFeedEventHost host = new ChangeFeedEventHost(
          hostName,
          documentCollectionLocation,
          leaseCollectionLocation,
          feedOptions,
          feedHostOptions
);
await host.RegisterObserverAsync(docObserverFactory);
```

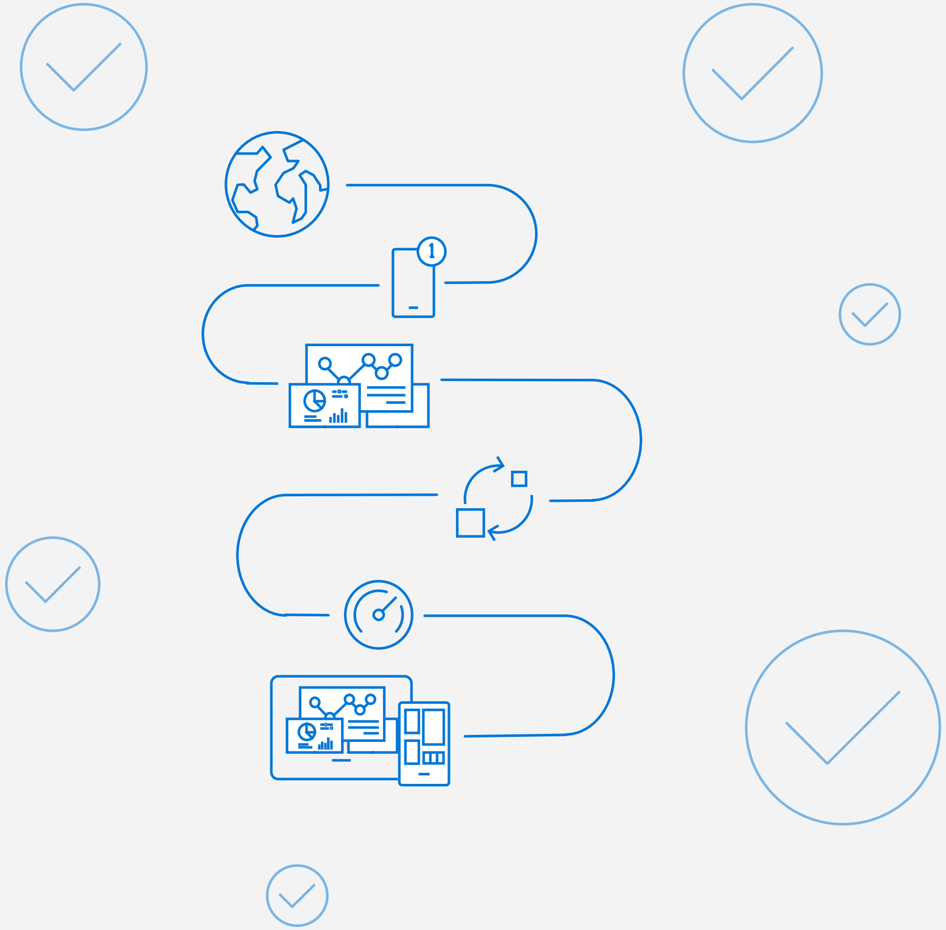
.NET

Troubleshooting

TROUBLESHOOTING

Using HTTP/REST response status codes and headers to diagnose and troubleshoot requests.

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



ANALYZING HTTP RESPONSES

RESPONSE STATUS CODES

When a request is unsuccessful, Azure Cosmos DB responds using well-defined HTTP status codes that can provide more detail into exactly why a specific request failed.

RESPONSE HEADERS

Azure Cosmos DB uses a variety of HTTP headers to offer insight into the result of requests, error conditions, and useful metadata to perform actions such as:

- Resume request
- Measure RU/s charge associated with request
- Access newly created resource directly.

HTTP response codes	
2xx	Success
4xx	Client Errors
5xx	Server Errors

RESPONSE STATUS CODES

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

RESPONSE STATUS CODES

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

RESPONSE HEADERS

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

RESPONSE HEADERS

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

IDENTIFYING RATE LIMITING

HTTP RESPONSE STATUS CODE

A rate limited request will return a HTTP status code of **429 (Too Many Requests)**. This response indicates that the container has exceeded provisioned throughput limit.

HTTP RESPONSE HEADER

A **rate limited** request will also have a **x-ms-retry-after-ms** header. This header gives the number of milliseconds your application should wait before retrying the current request.

AUTOMATIC RETRY ON THROTTLE

The SDK automatically retries any throttled requests. This can **potentially create a long-running client-side method** that is attempting to retry throttled requests.

LOGGING

WHAT IS LOGGED BY AZURE DIAGNOSTIC LOGS

All authenticated backend requests across all protocols and APIs

- Includes failed requests

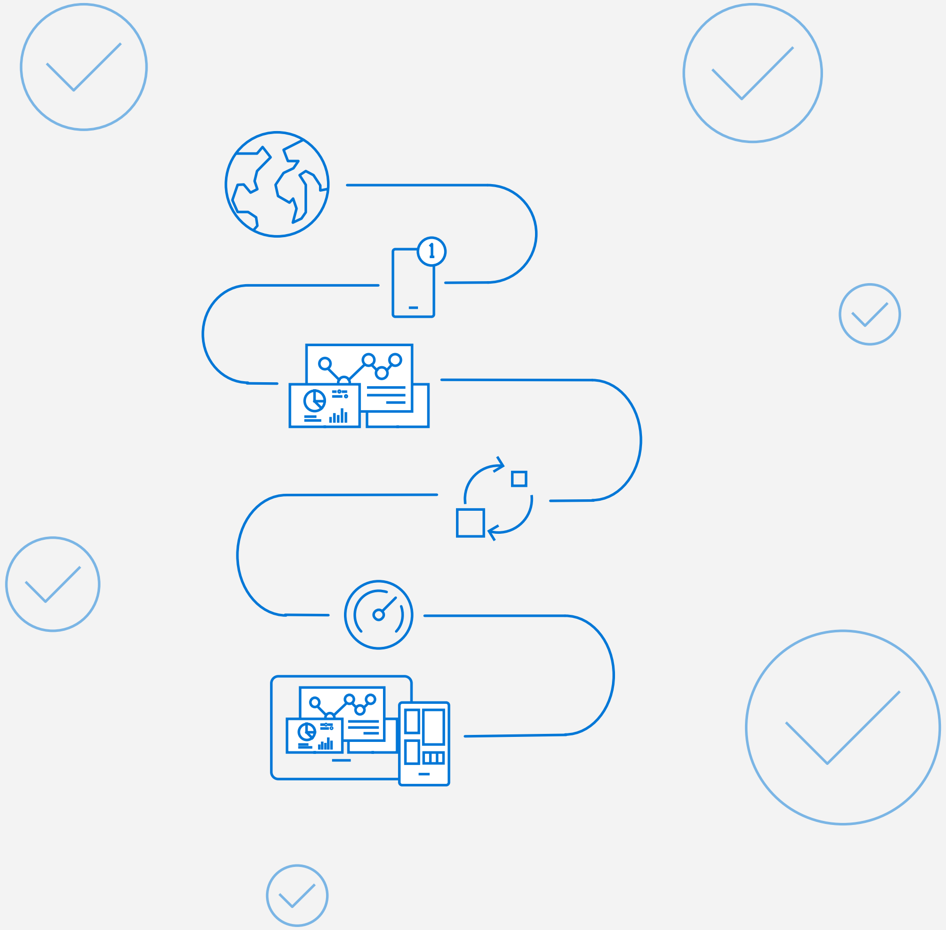
Database operations

- 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:

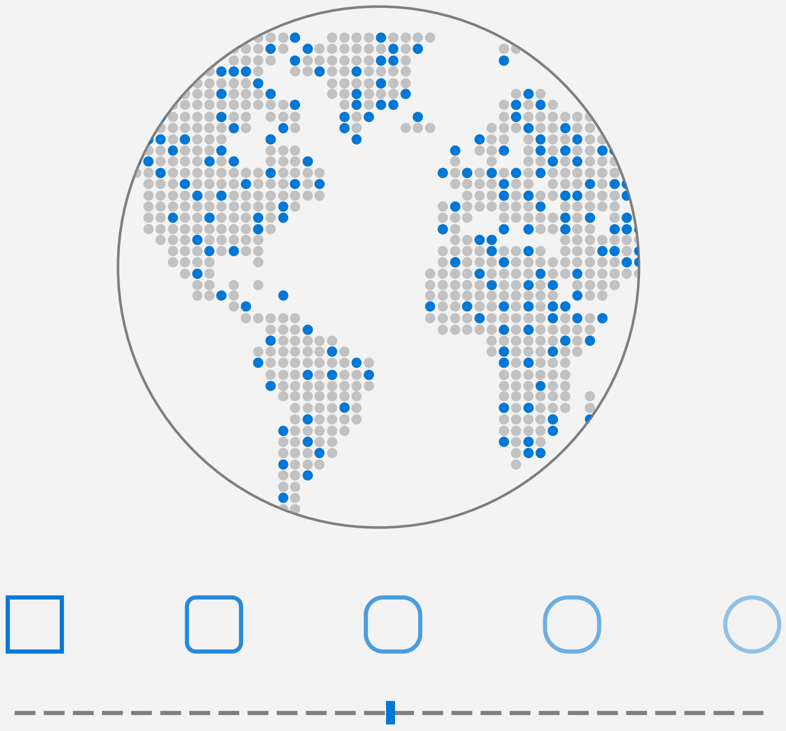
- Search logs using the expanded Log Analytics query language
- Visualize the results of a query as a graph
- Pin a graph visualization of a query to your Azure portal dashboard

Modeling & Planning

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.

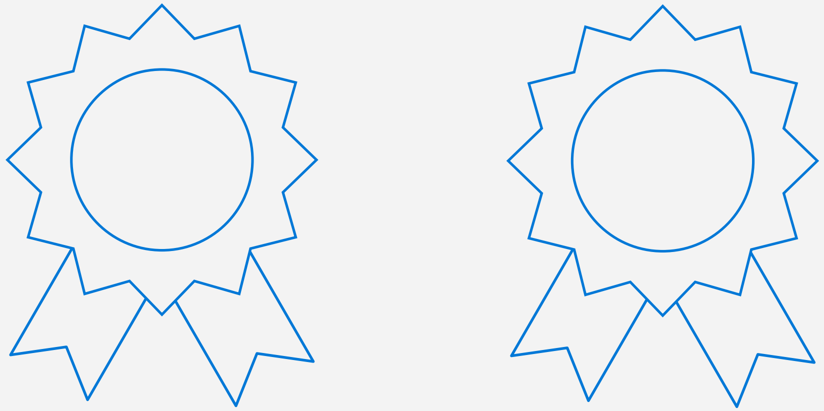
```
{
        "id": "08259",
        "flightCode": "3754",
        "pilot": [{ "id": "EBAMAO", “name": "Hailey Nelson“ }]

},
{
        "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)

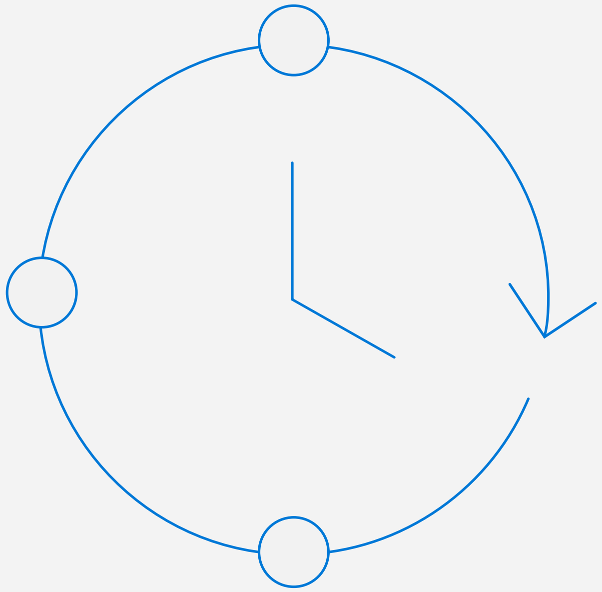
AUTOMATICALLY PURGE DATA

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.



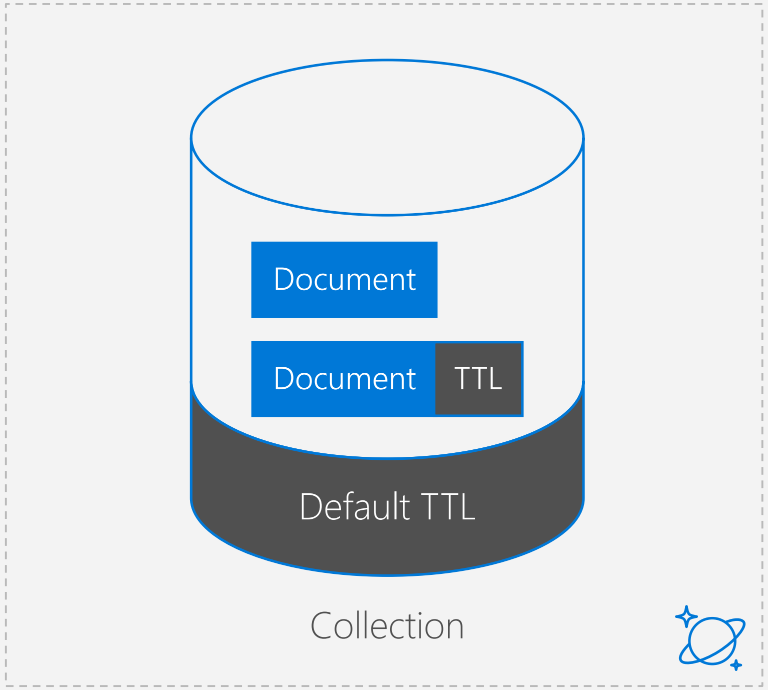
EXPIRING RECORDS USING TIME-TO-LIVE

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.



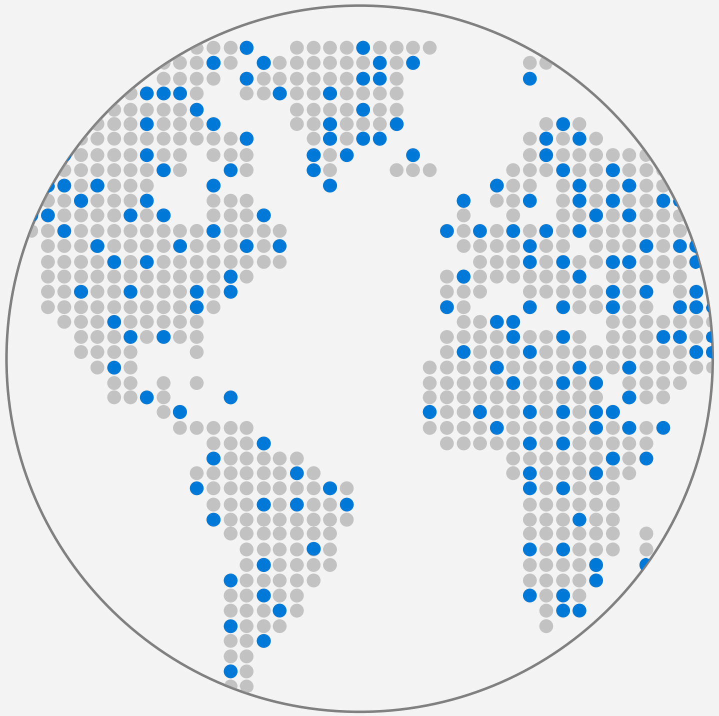
TURNKEY GLOBAL DISTRIBUTION

High Availability

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

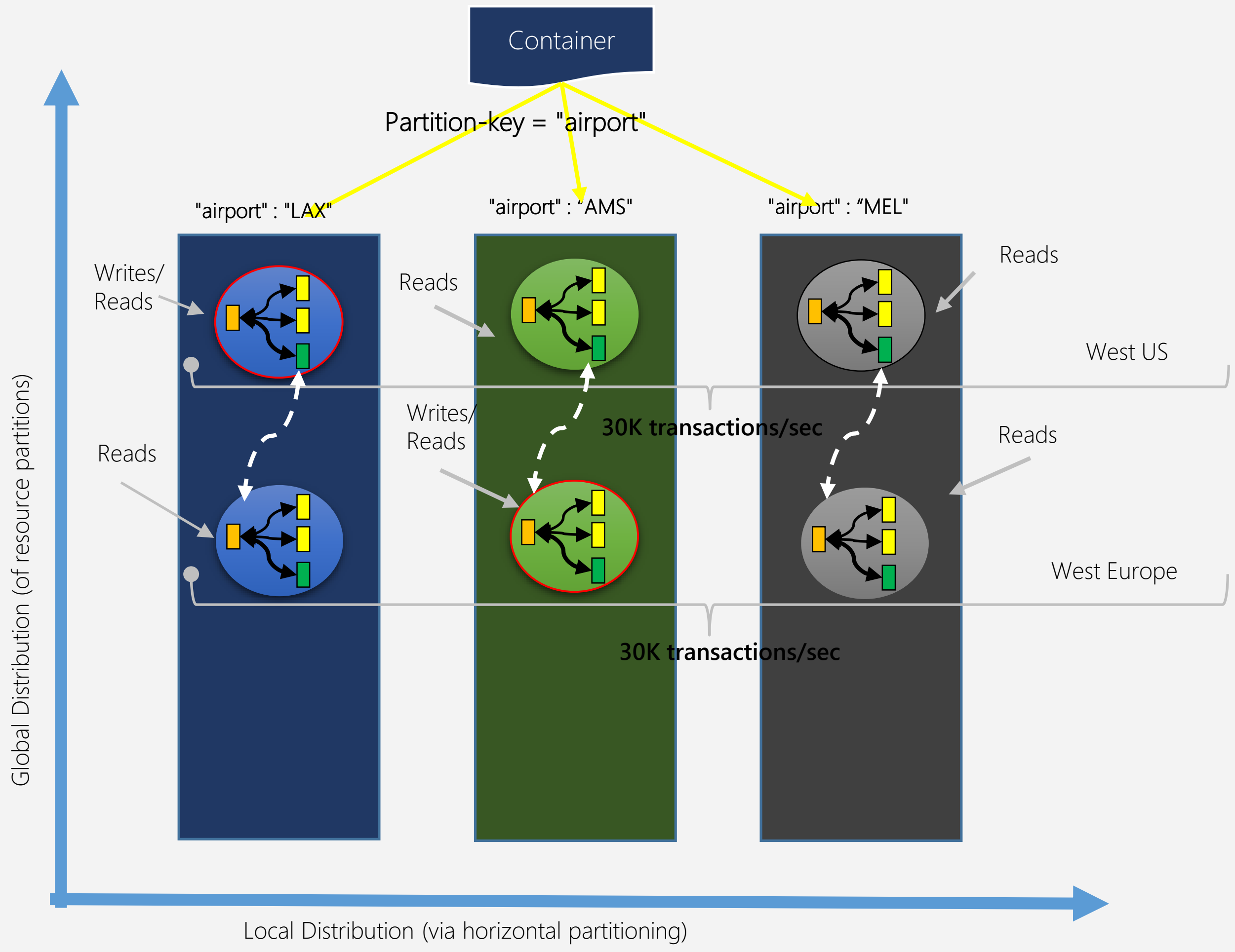
Low Latency (anywhere in the world)

- Packets cannot move 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

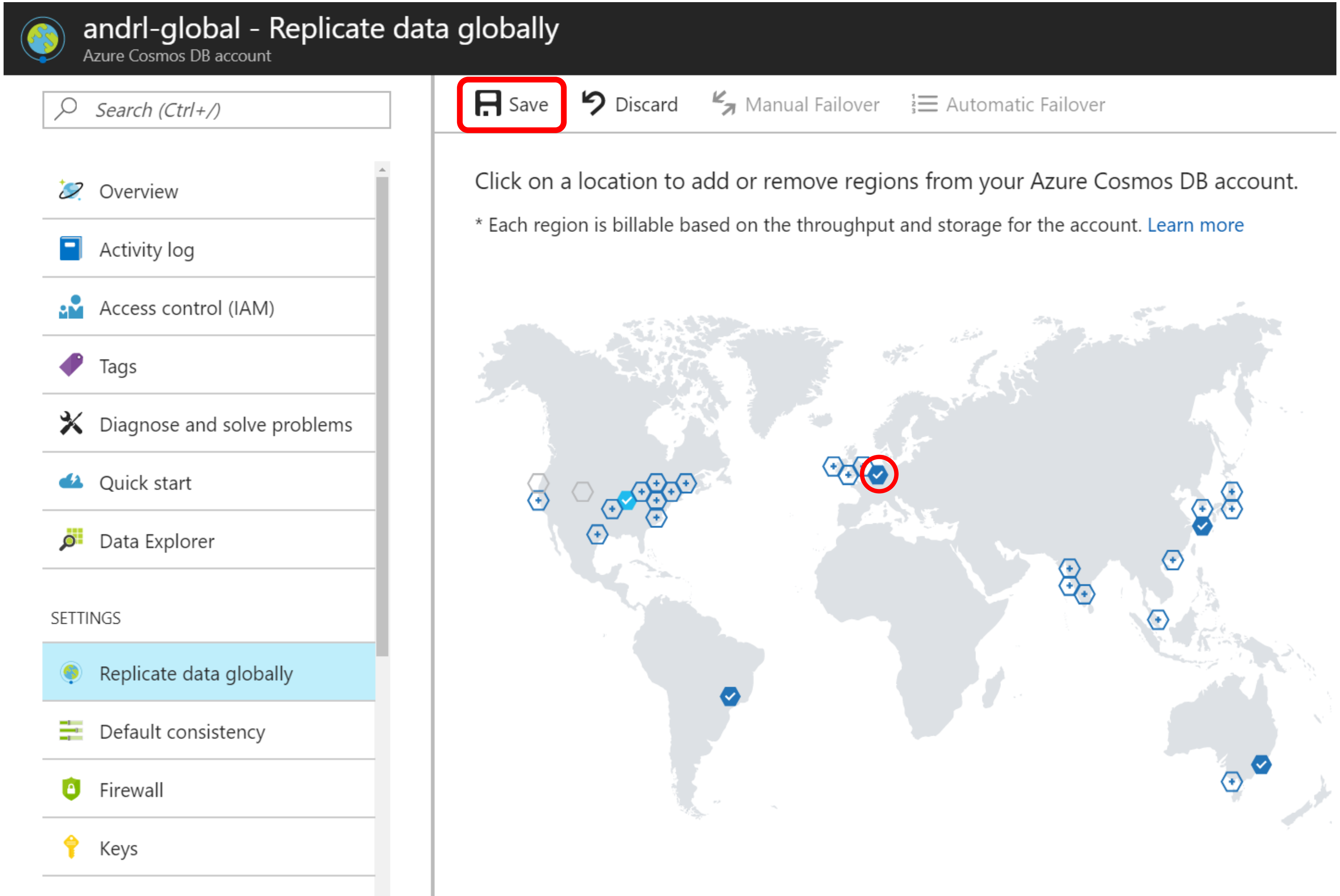


TURNKEY GLOBAL DISTRIBUTION

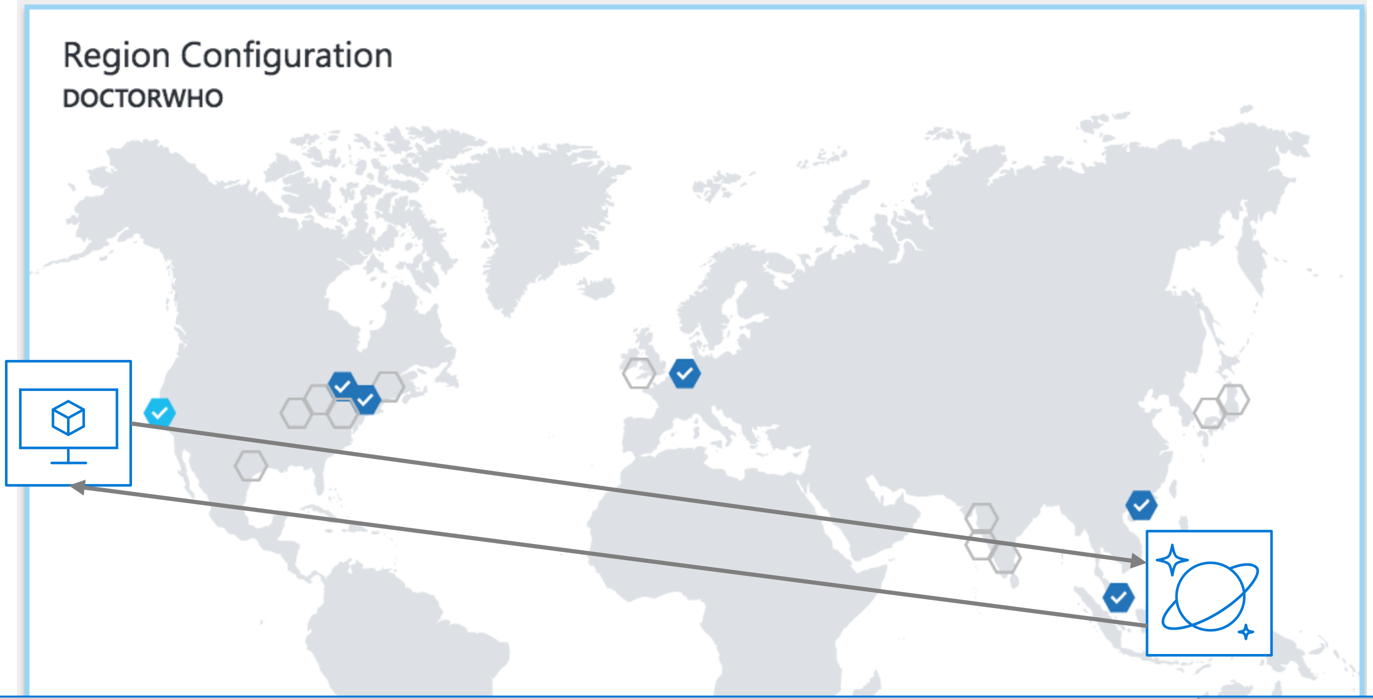
- Automatic and transparent replication worldwide
- Each partition hosts a replica set per region
- Customers can test end to end application availability by programmatically simulating failovers
- All regions are hidden behind a single global URI with multi-homing capabilities
- Customers can dynamically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



REPLICATING DATA GLOBALLY

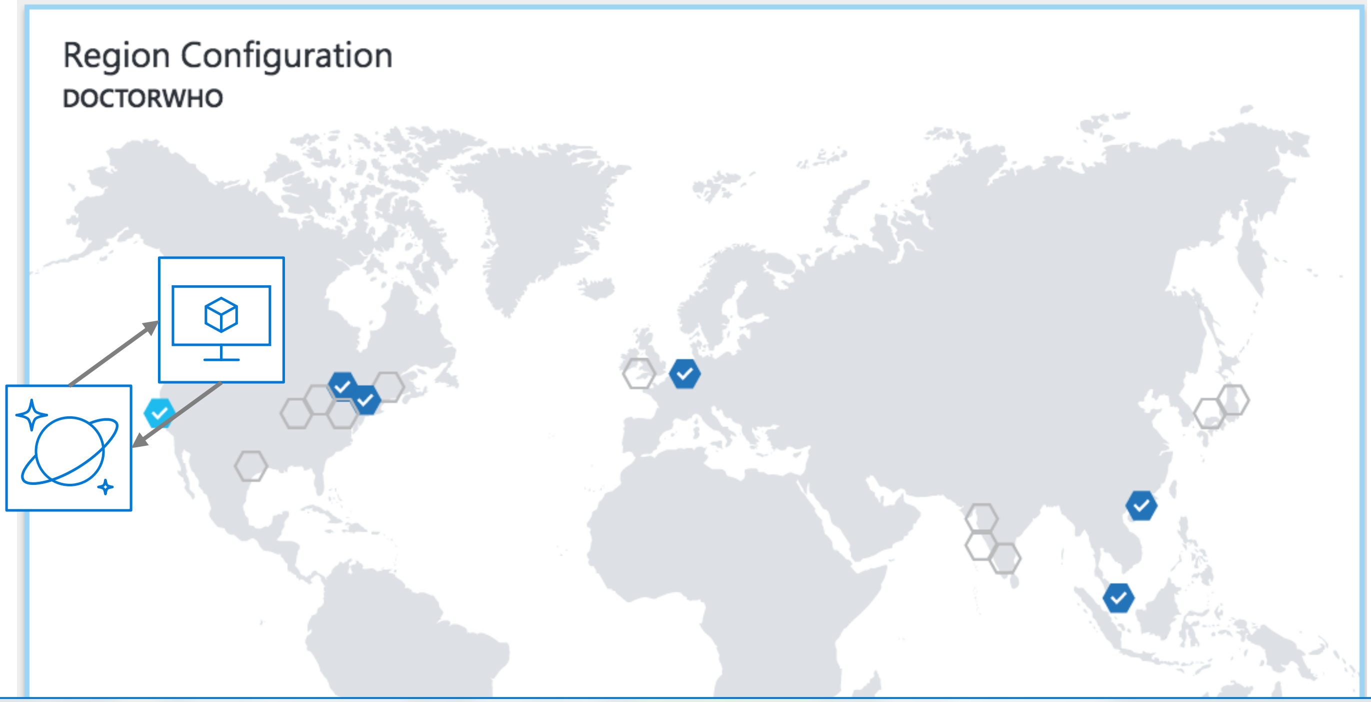


REPLICATING DATA GLOBALLY



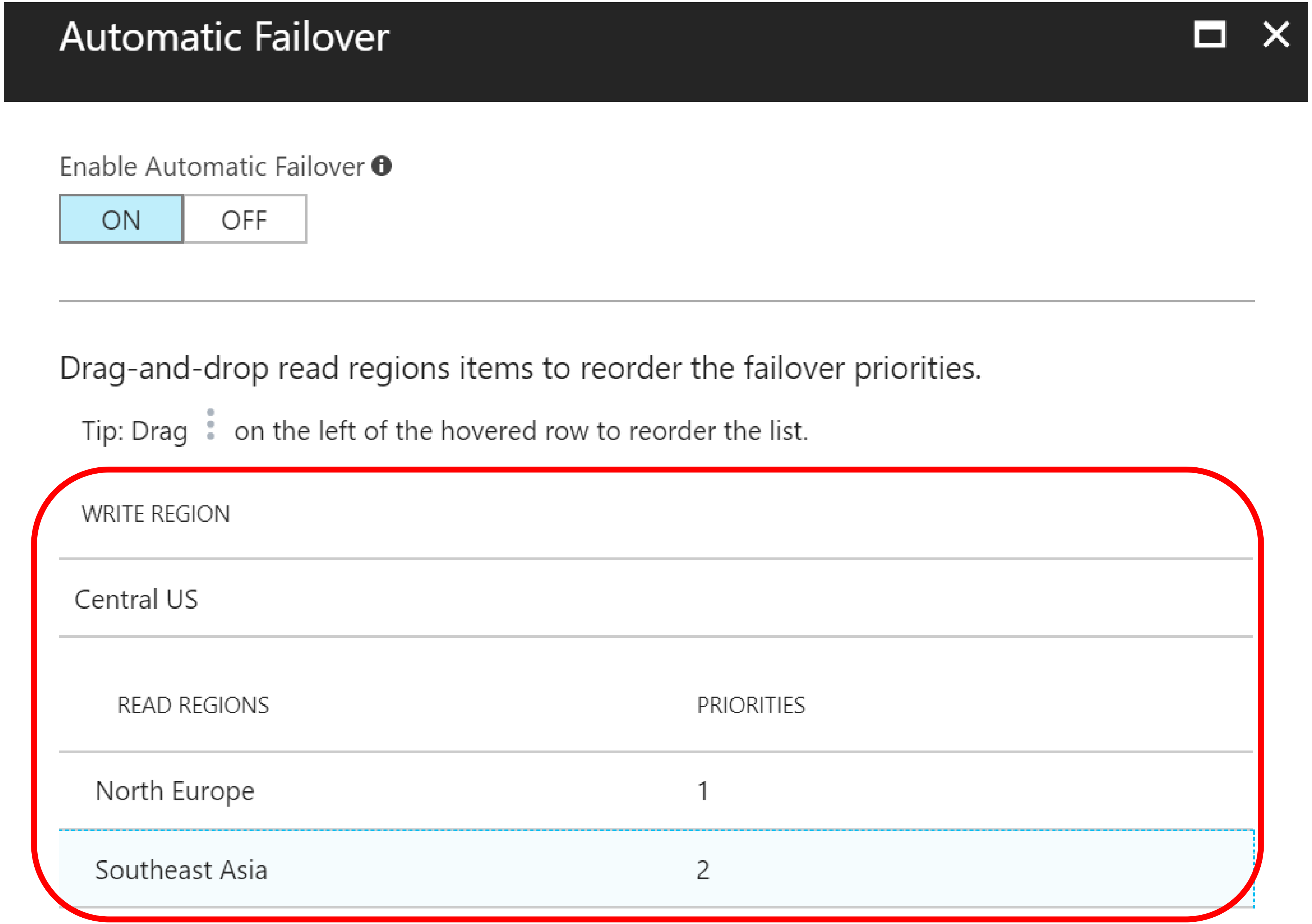

```
...rkspace/docdb/docdb — amypond@blink: ~/docdb — ssh amypond@104.42.108.173 -p 22
[amypond@blink:~/docdb$ node testQ2.js
210.788804 milliseconds, 2 RUs, ActivityId: 9025eac6-eb74-4a07-94cf-f2383caffbb3
178.825773 milliseconds, 2 RUs, ActivityId: ea53b736-b629-4290-9cdf-cf80c6139461
178.839173 milliseconds, 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
```

AUTOMATIC FAILOVER



AUTOMATIC FAILOVER

```
DocumentClient client;
ConnectionPolicy policy = new ConnectionPolicy()
{
    ConnectionMode = ConnectionMode.Direct,
    ConnectionProtocol = Protocol.Tcp
};
policy.PreferredLocations.Add(LocationNames.CentralUS); //first preference
policy.PreferredLocations.Add(LocationNames.SoutheastAsia); //second preference
policy.PreferredLocations.Add(LocationNames.NorthEurope); //third preference

client = new DocumentClient(new Uri(endpointUri), PrimaryKey, connectionPolicy: policy);
```

.NET

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

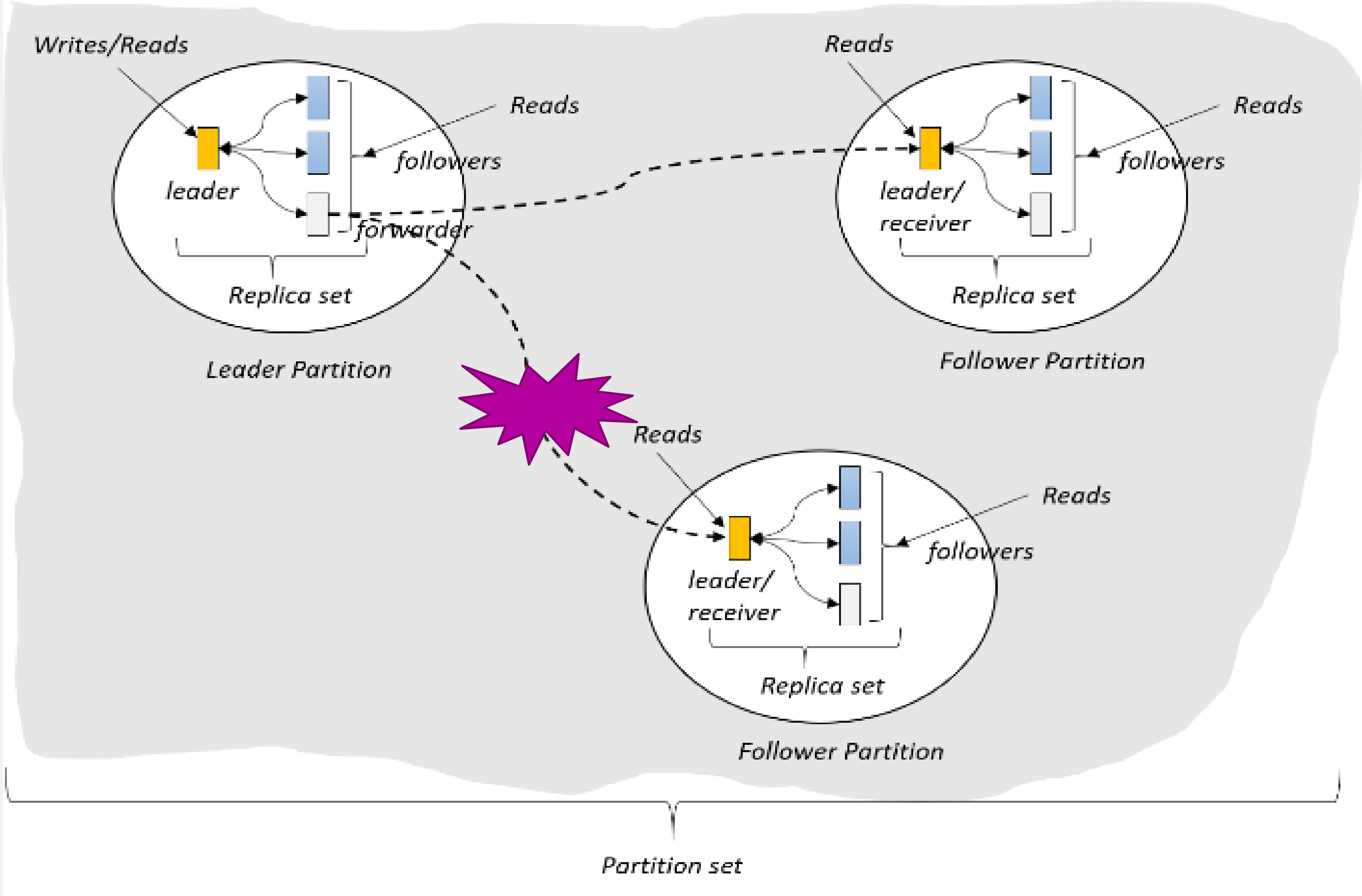
North Europe

- [x] I understand and agree to trigger a failover on my current Write Region.

OK

Consistency

BREWER'S CAP THEOREM



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

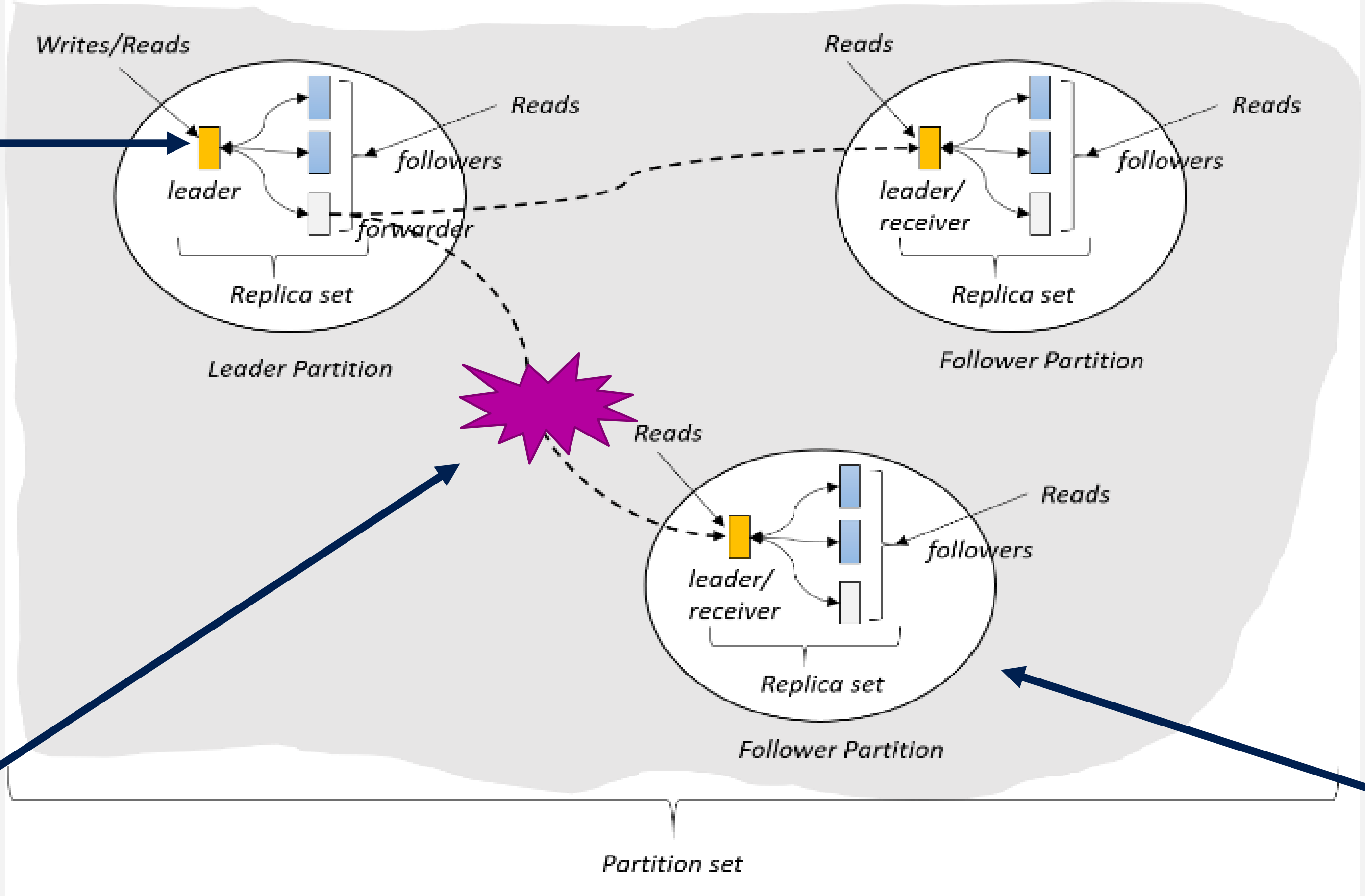
- Consistency
- Availability
- Partition Tolerance

CONSISTENCY

(West US)

Value = ~~5~~ 6

Update 5 => 6



(East US)

Value = ~~5~~ 6

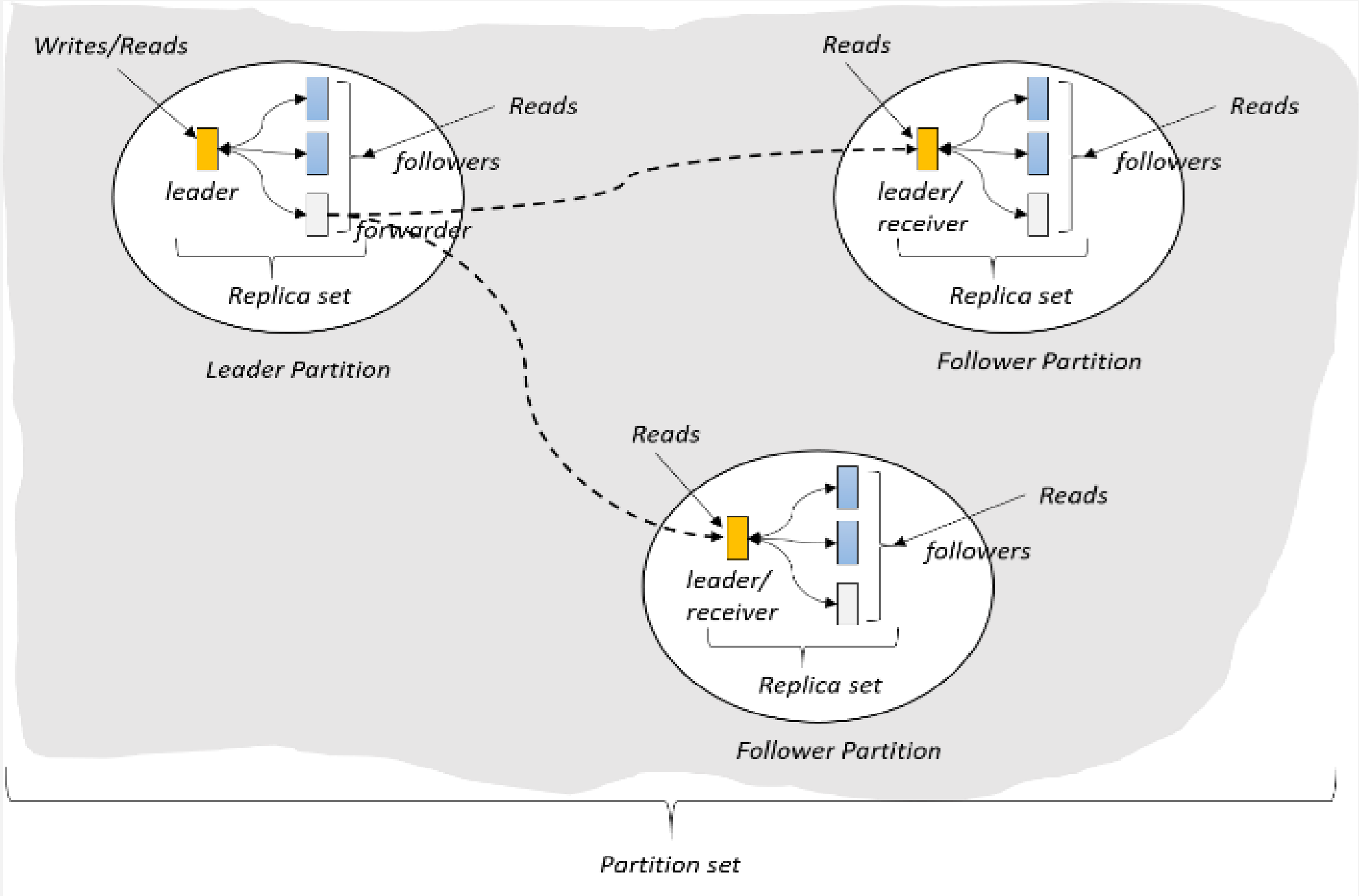
(North Europe)

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



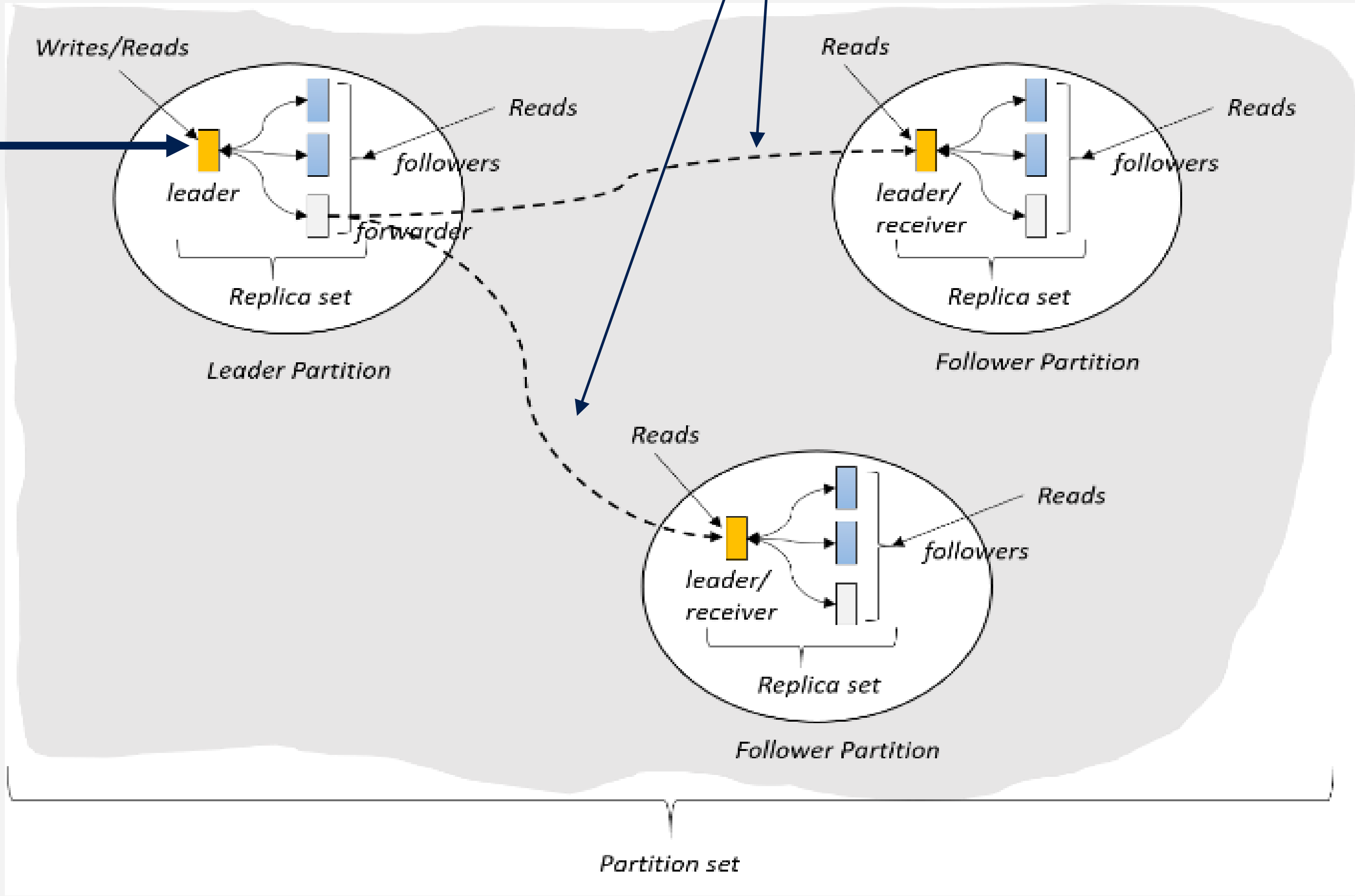
In the case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency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

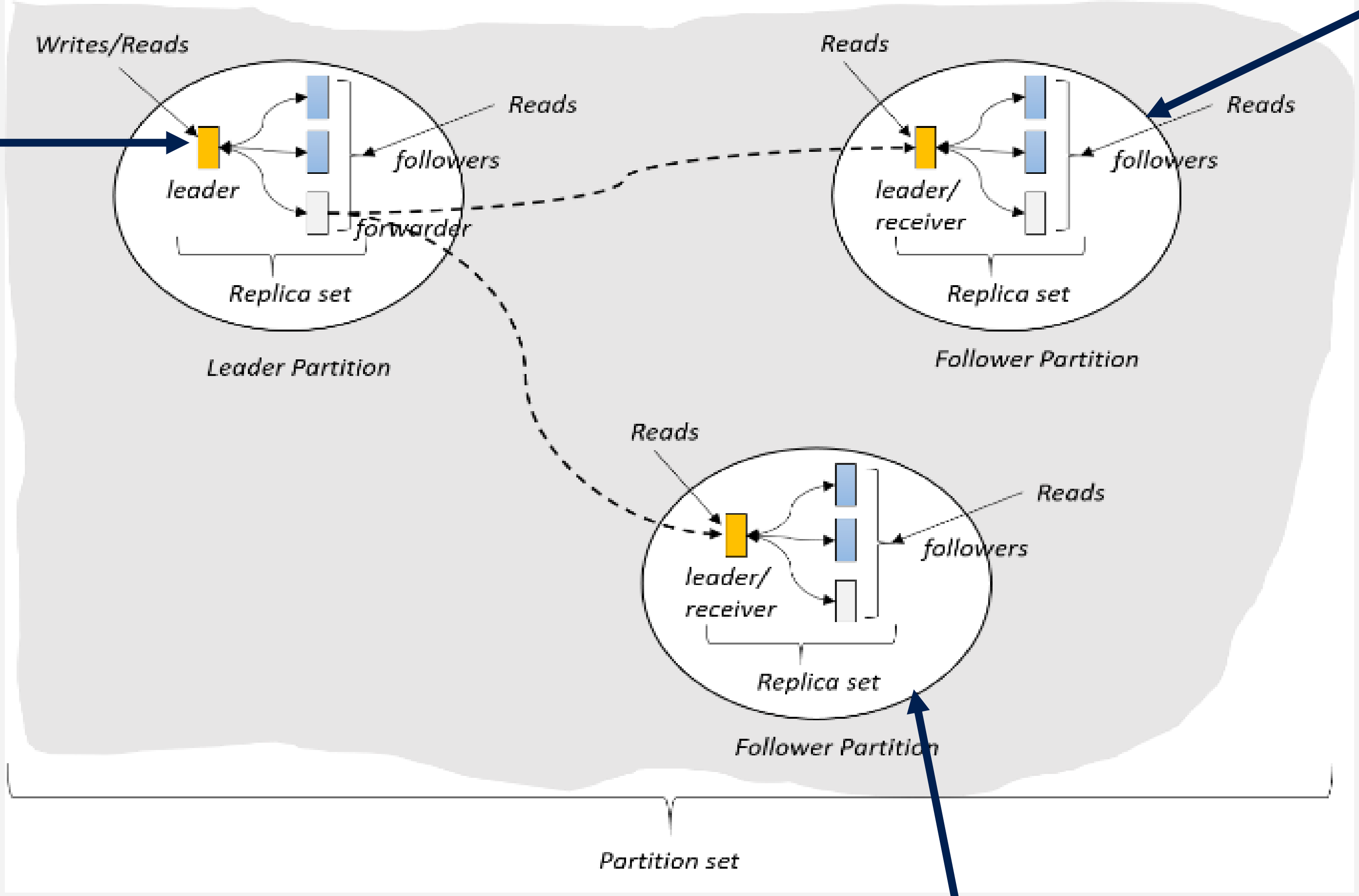
Value = 5

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

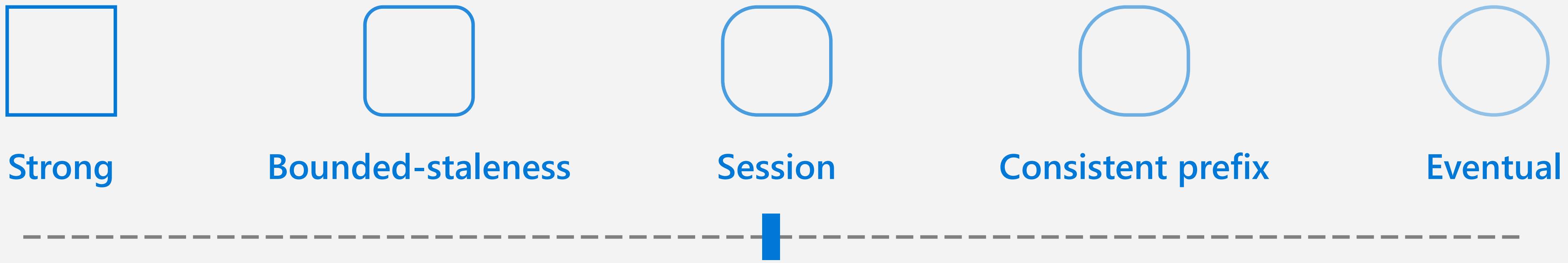
Overridable on a per-request basis

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



CONSISTENCY MODELS - BREAKDOWN

Consistency Level	Guarantees
Strong	Linearizability (once operation is complete, it will be visible to all)
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), while preserving 99.99% availability and low latency.
Session	Consistent Prefix. Within a session: monotonic reads, monotonic writes, read-your-writes, write-follows-reads Predictable consistency for a session, high read throughput + low latency
Consistent Prefix	Reads will never see out of order writes (no gaps).
Eventual	Potential for out of order reads. Lowest cost for reads of all consistency levels.

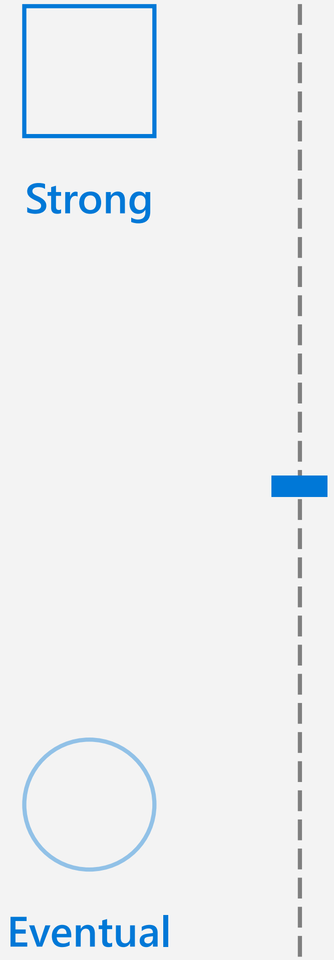
DEMYSTIFYING CONSISTENCY MODELS

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.



STRONG CONSISTENCY

- Strong consistency offers a linearizability guarantee with the reads guaranteed to return the most recent version of an item.
- Strong consistency guarantees that a write is only visible after it is committed durably by the majority quorum of replicas.
- A client is always guaranteed to read the latest acknowledged write.
- The cost of a read operation (in terms of request units consumed) with strong consistency is higher than session and eventual, but the same as bounded staleness.

EVENTUAL CONSISTENCY

- Eventual consistency guarantees that in absence of any further writes, the replicas within the group eventually converge.
- Eventual consistency is the weakest form of consistency where a client may get the values that are older than the ones it had seen before.
- Eventual consistency provides the weakest read consistency but offers the lowest latency for both reads and writes.
- The cost of a read operation (in terms of RUs consumed) with the eventual consistency level is the lowest of all the Azure Cosmos DB consistency levels.

DEMYSTIFYING CONSISTENCY MODELS

Bounded-staleness

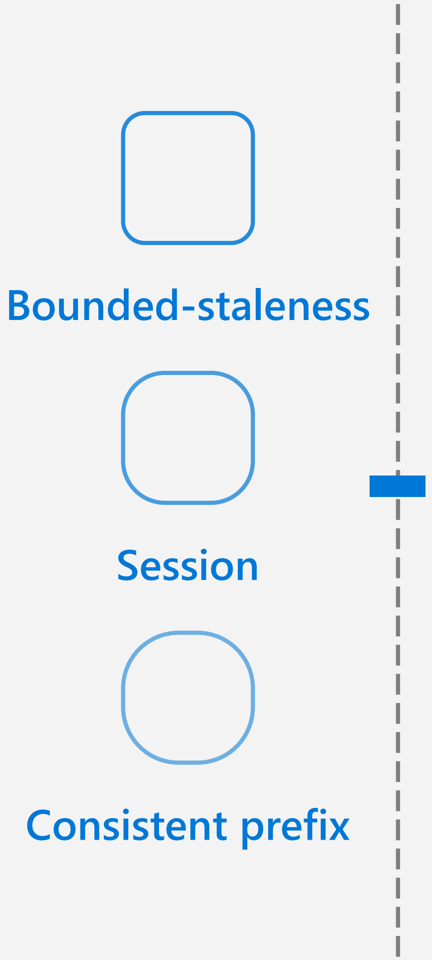
Consistent prefix. Reads lag behind writes by at most **k** prefixes or **t** interval. Similar properties to strong consistency except within staleness window.

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.



BOUNDED STALENESS CONSISTENCY

- Bounded staleness consistency guarantees that the reads may lag behind writes by at most K versions or prefixes of an item or t time-interval.
- Bounded staleness offers total global order except within the "staleness window." The monotonic read guarantees exist within a region both inside and outside the "staleness window."
- Bounded staleness provides a stronger consistency guarantee than session, consistent-prefix, or eventual consistency.
- The cost of a read operation (in terms of RUs consumed) with bounded staleness is higher than session and eventual consistency, but the same as strong consistency.

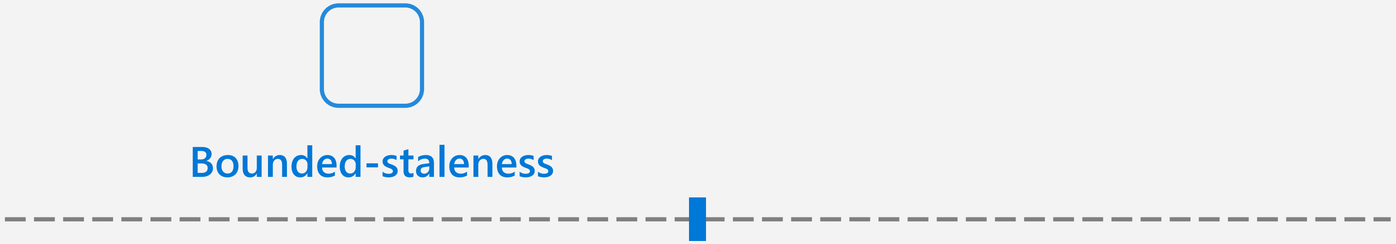
SESSION CONSISTENCY

- Unlike the global consistency models offered by strong and bounded staleness consistency levels, session consistency is scoped to a client session.
- Session consistency is ideal for all scenarios where a device or user session is involved since it guarantees monotonic reads, monotonic writes, and read your own writes (RYW) guarantees.
- Session consistency provides predictable consistency for a session, and maximum read throughput while offering the lowest latency writes and reads.
- The cost of a read operation (in terms of RUs consumed) with session consistency level is less than strong and bounded staleness, but more than eventual consistency.

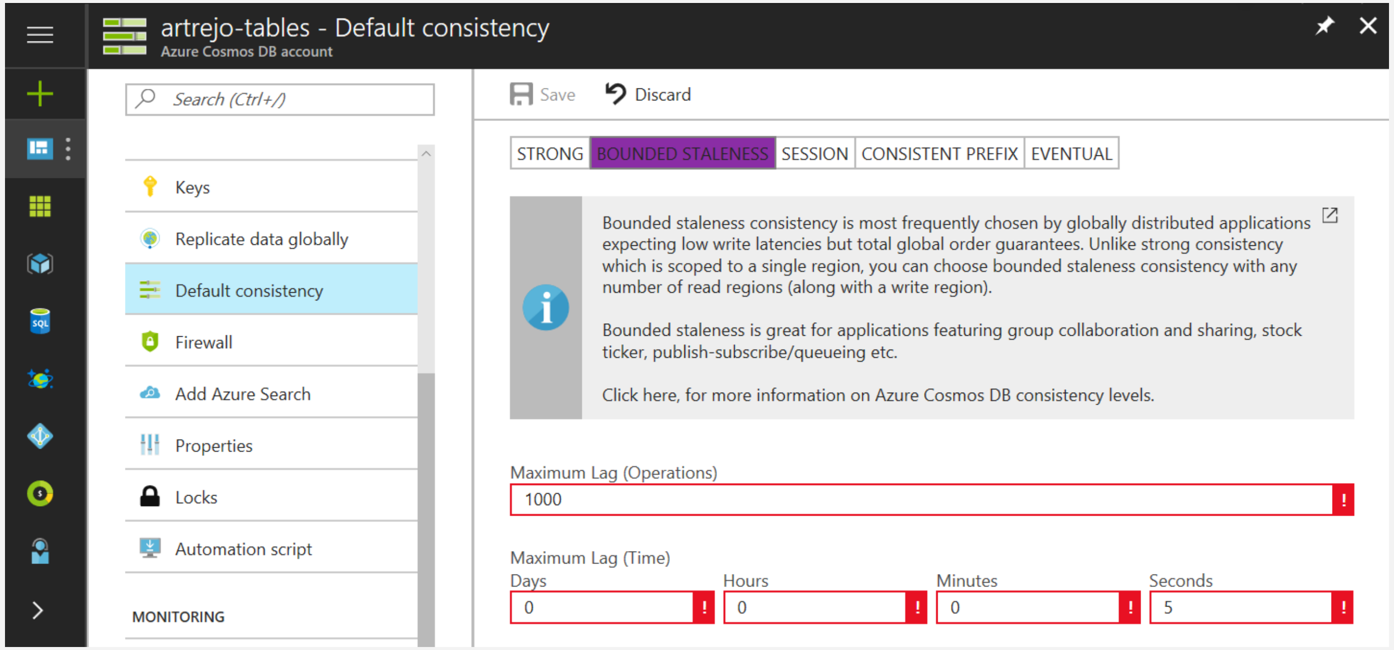
CONSISTENT PREFIX CONSISTENCY

- Consistent prefix guarantees that in absence of any further writes, the replicas within the group eventually converge.
- Consistent prefix guarantees that reads never see out of order writes. If writes were performed in the order **A**, **B**, **C**, then a client sees either **A**, **A**,**B**, or **A**,**B**,**C**, but never out of order like **A**,**C** or **B**,**A**,**C**.

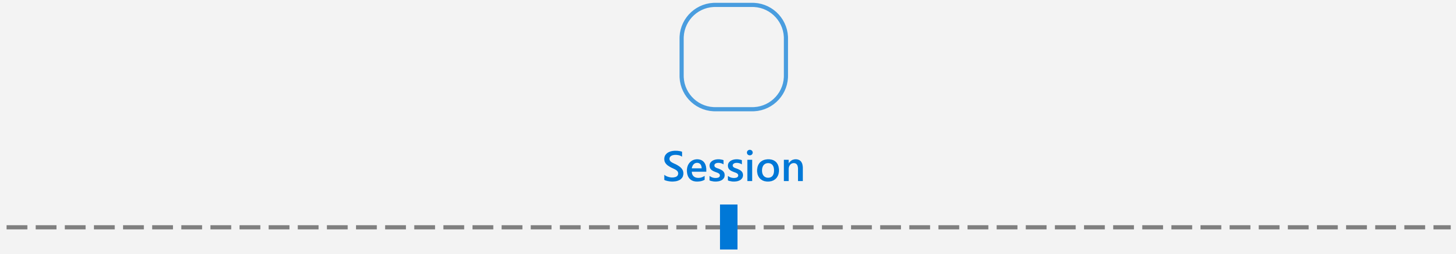
BOUNDED STALENESS IN THE PORTAL



BOUNDS ARE SET SERVER-SIDE VIA THE AZURE PORTAL



SESSION CONSISTENCY IN CODE



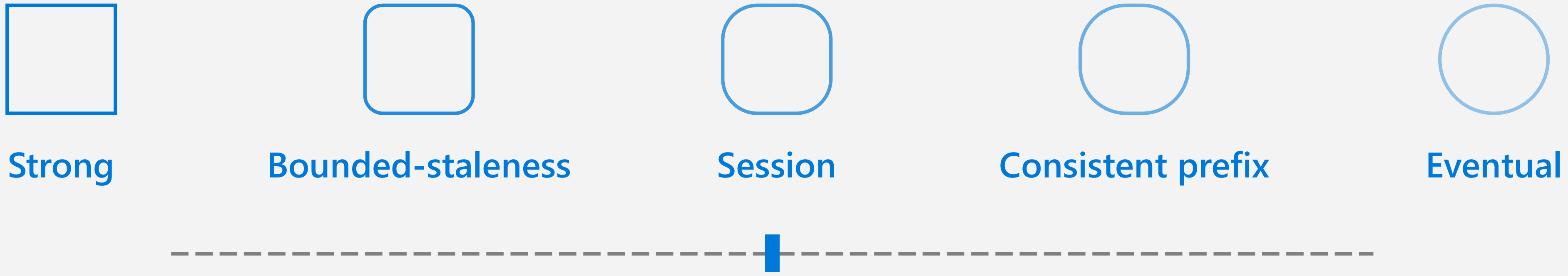
SESSION IS CONTROLLED USING A “SESSION TOKEN”

- Session tokens are automatically cached by the Client SDK
- Can be pulled out and used to override other requests (to preserve session between multiple clients)

```
string sessionToken;

using (DocumentClient client = new DocumentClient(new Uri(""), ""))
{
  ResourceResponse<Document> response = client.CreateDocumentAsync(
      collectionLink,
      new { id = "an id", value = "some value" }
  ).Result;
  sessionToken = response.SessionToken;
}
```

```
using (DocumentClient client = new DocumentClient(new Uri(""),
""))
{
  ResourceResponse<Document> read = client.ReadDocumentAsync(
      documentLink,
      new RequestOptions { SessionToken = sessionToken }
  ).Result;
}
```

RELAXING CONSISTENCY IN CODE



CONSISTENCY CAN BE RELAXED ON A PER-REQUEST BASIS

```
client.ReadDocumentAsync(
    documentLink,
    new RequestOptions { ConsistencyLevel = ConsistencyLevel.Eventual }
);
```