Azure Cosmos DB

Best Practices

AGENDA

Azure Cosmos DB Overview

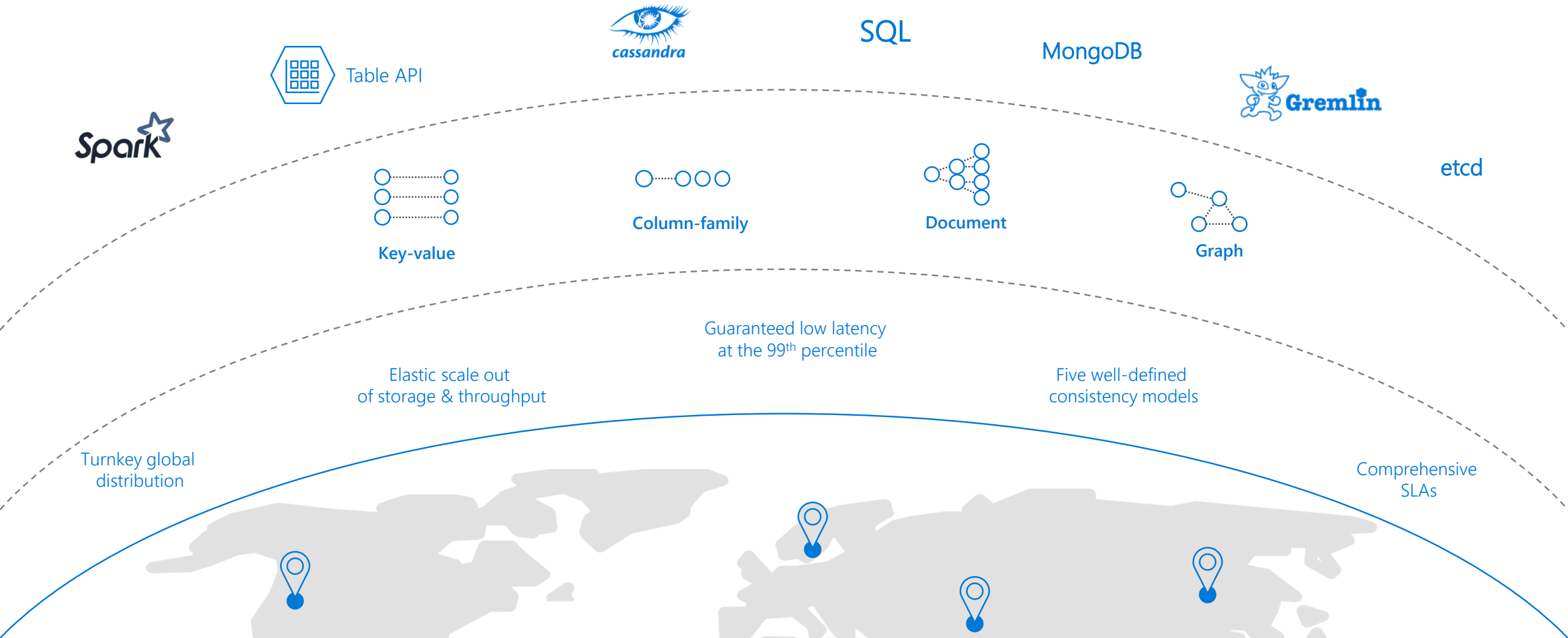
Partitioning

Querying

Programming

AZURE COSMOS DB

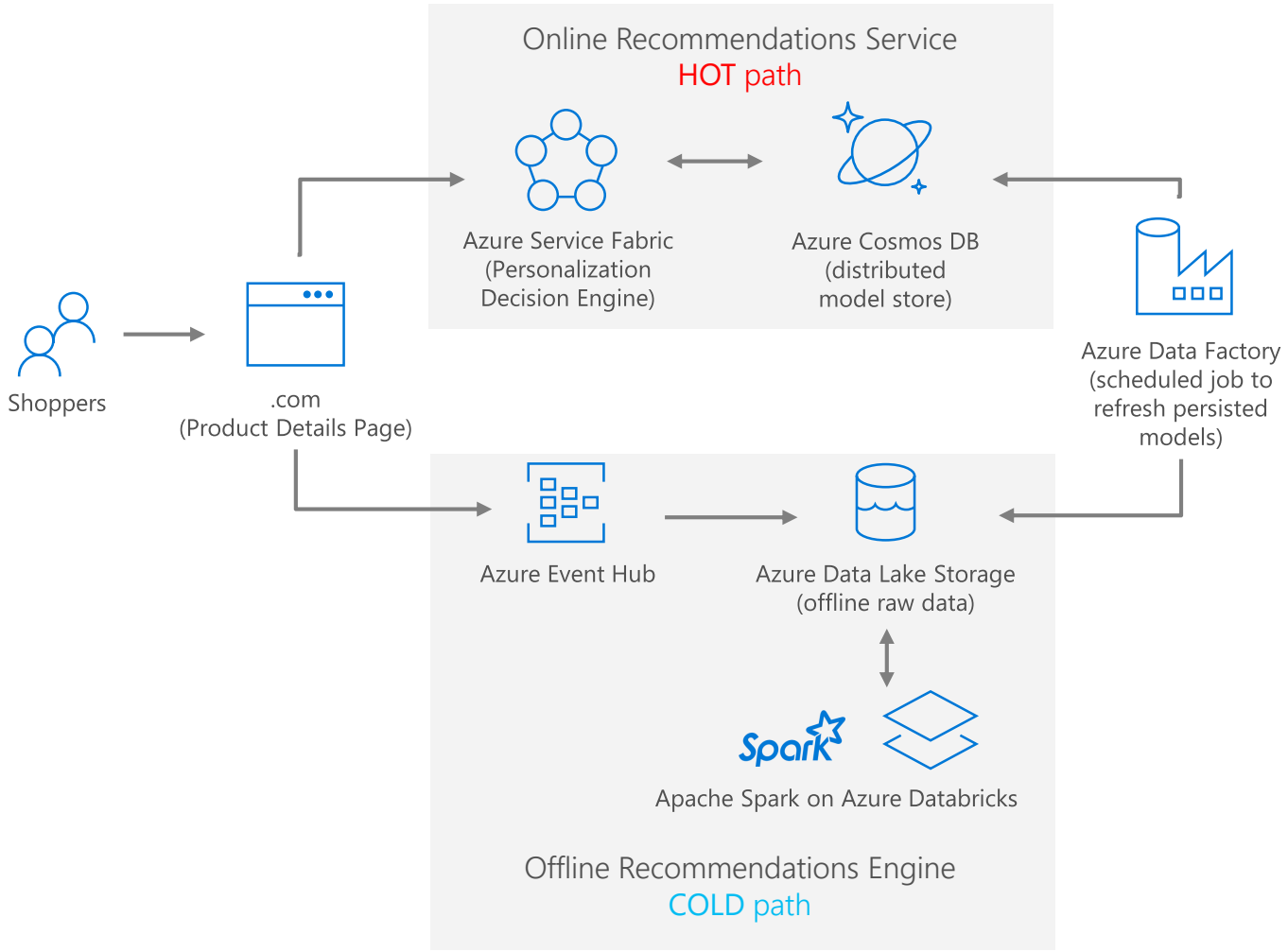
A globally distributed, massively scalable, multi-model database service



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

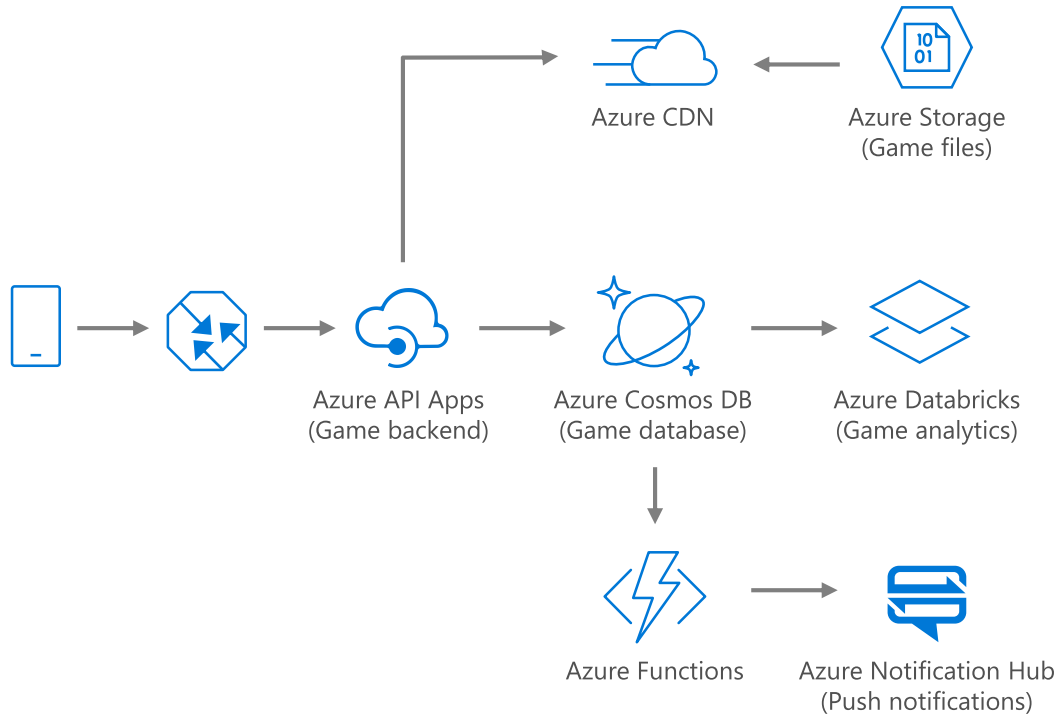


asos

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

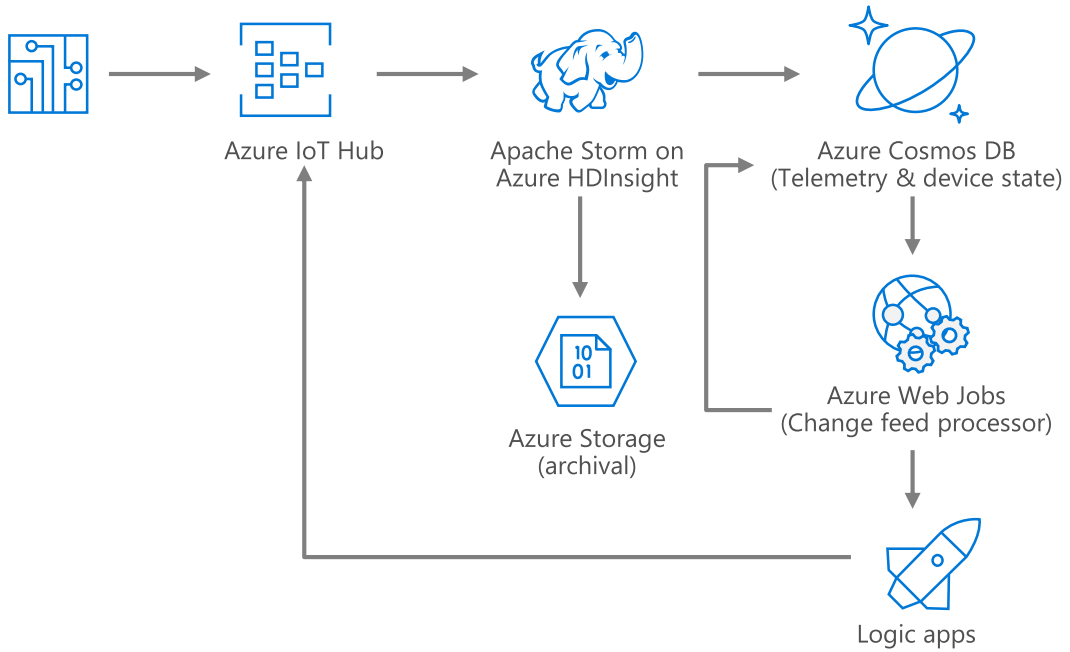


NEXT GAMES
Domino's
jet
XBOX LIVE

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

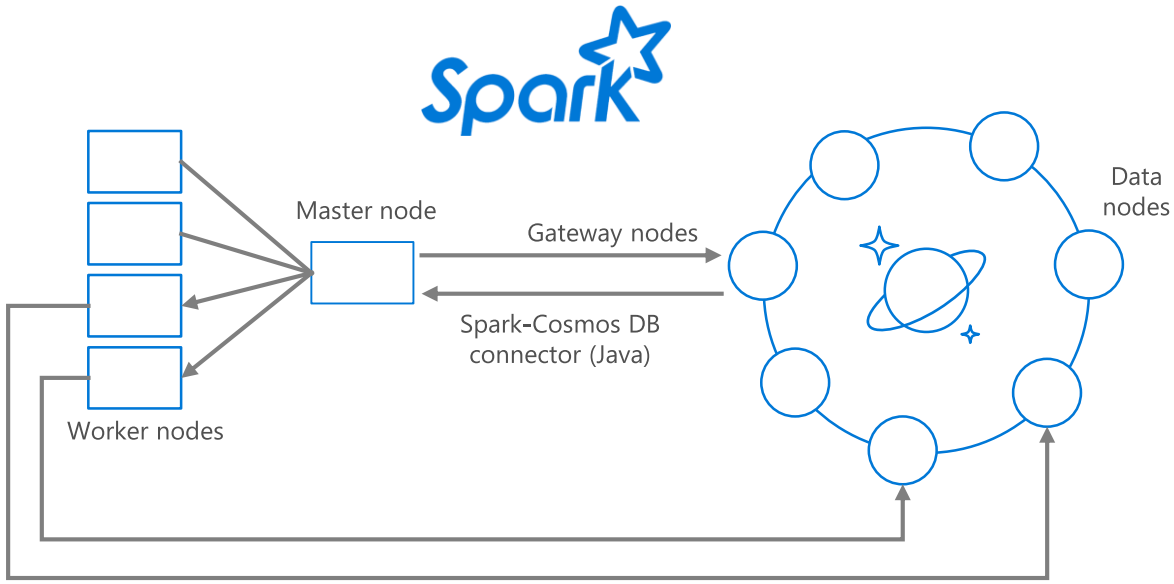


TOYOTA Honeywell LG CNS Johnson Controls

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

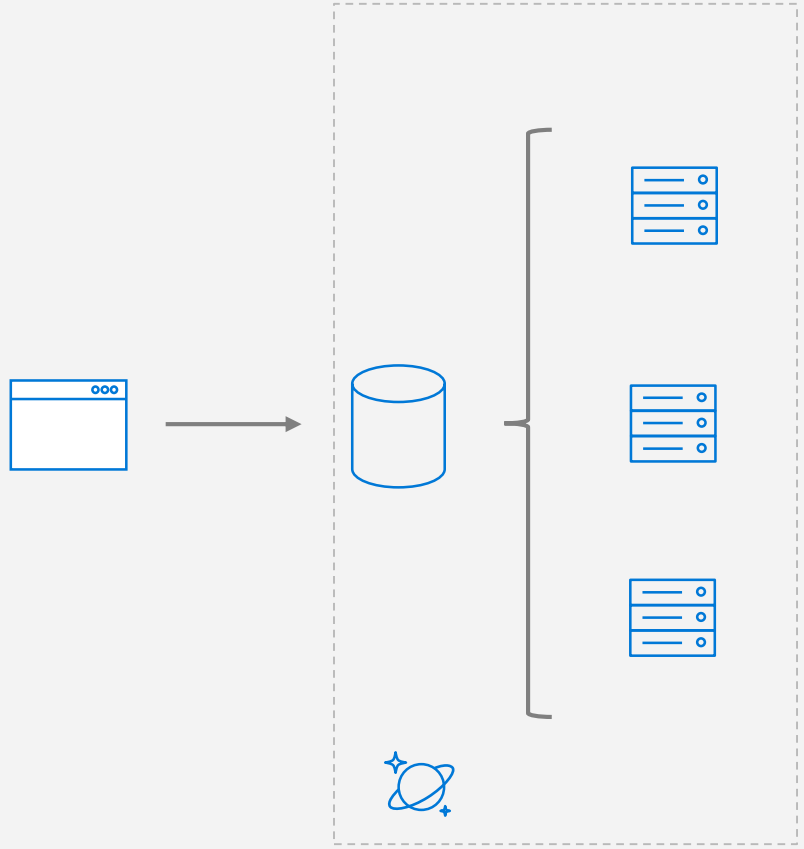


Affinio

PARTITIONING

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

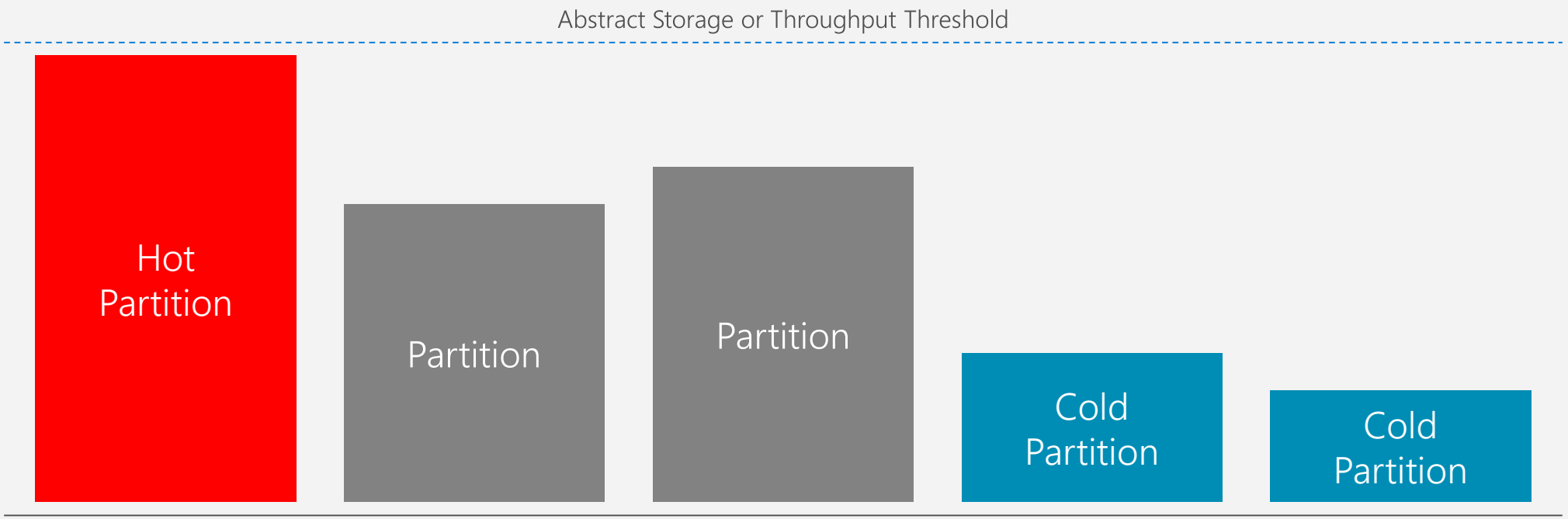
This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.



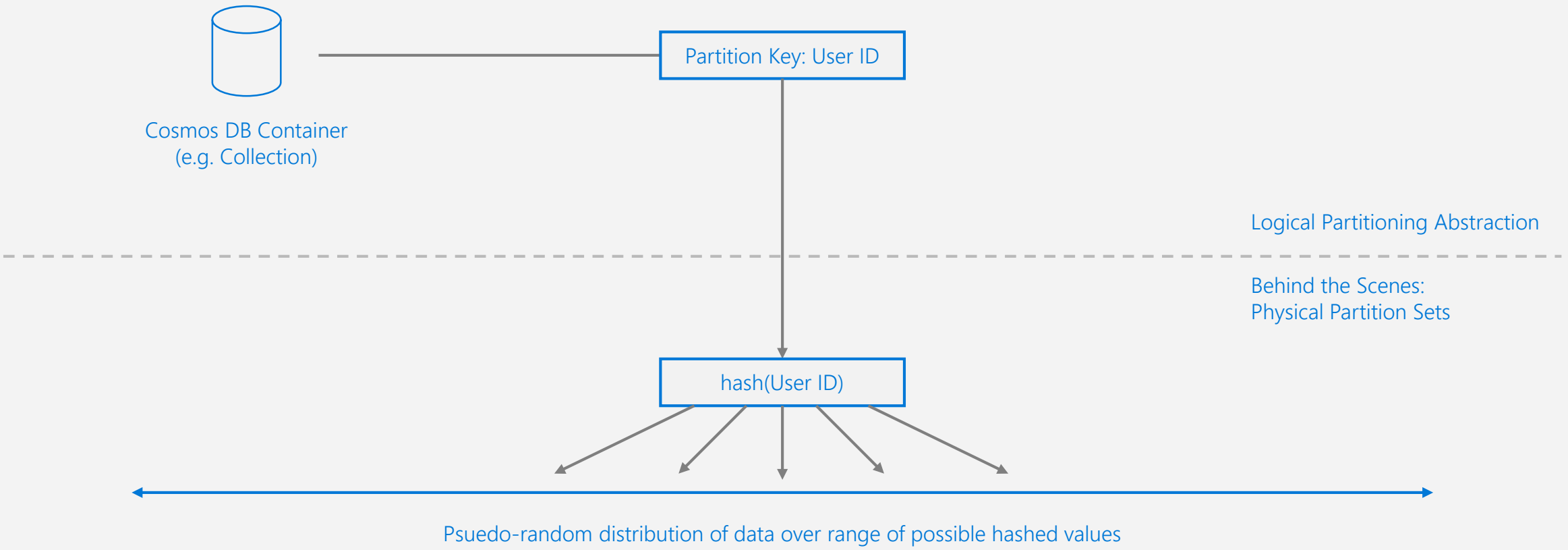
HOT/COLD PARTITIONS

PARTITION USAGE CAN VARY OVER TIME

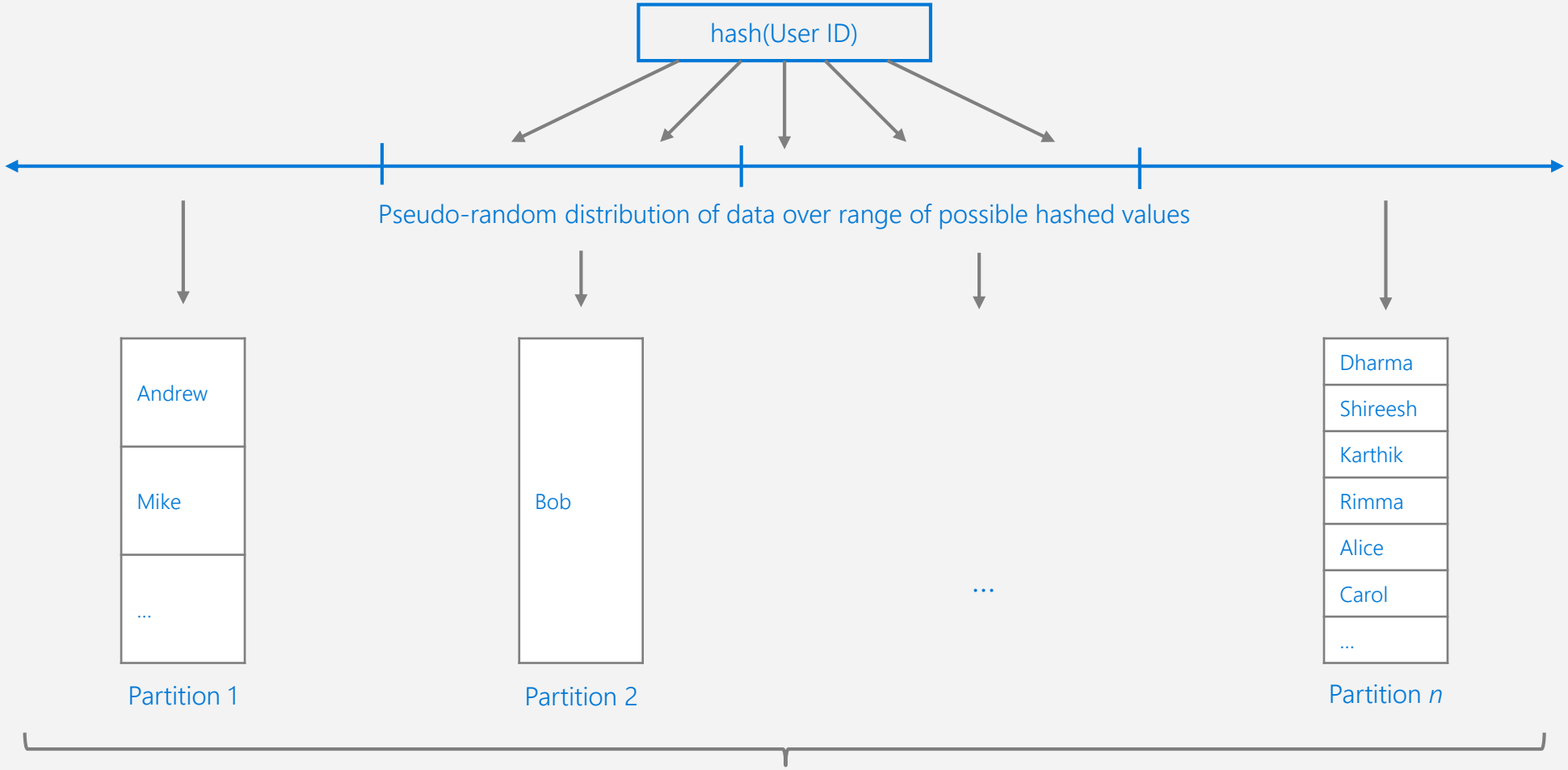
Partitions that are approaching thresholds are referred to as **hot**. Partitions that are underutilized are referred to as **cold**.



PARTITIONS

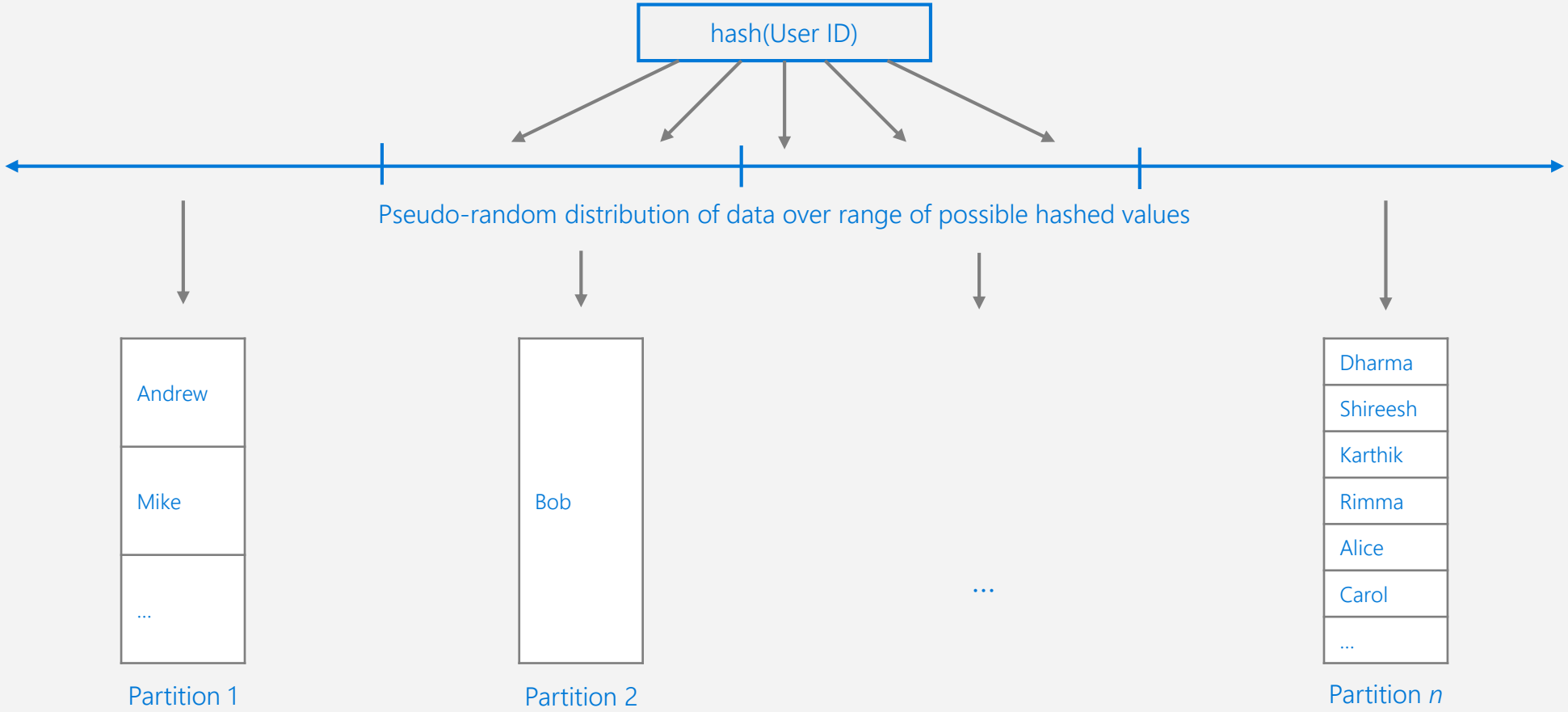

Partition Key: User ID
Cosmos DB Container
(e.g. Collection)
Logical Partitioning Abstraction
Behind the Scenes:
Physical Partition Sets
hash(User ID)
Psuedo-random distribution of data over range of possible hashed values

PARTITIONS



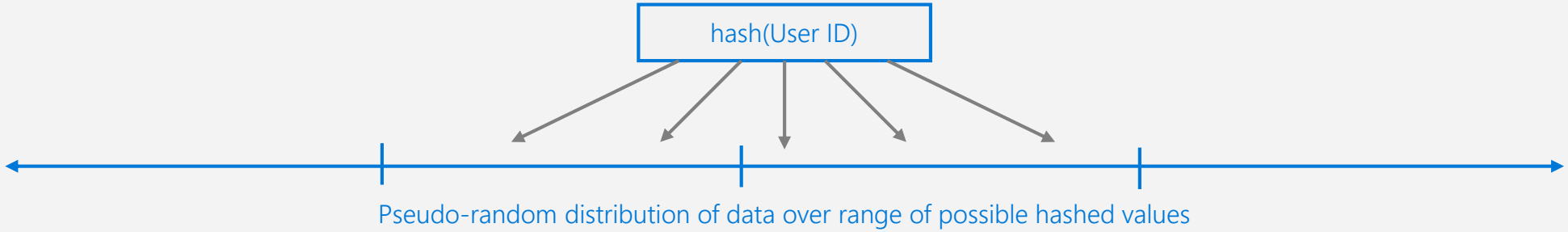
Frugal # of Partitions based on actual storage and throughput needs
(yielding scalability with low total cost of ownership)

PARTITIONS



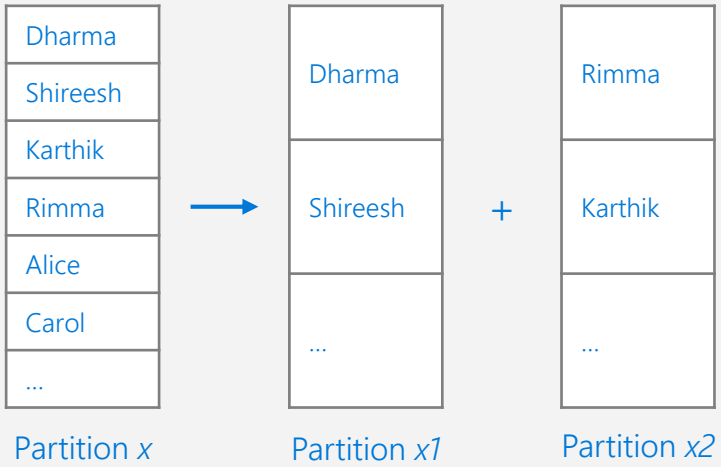
What happens when partitions need to grow?

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.



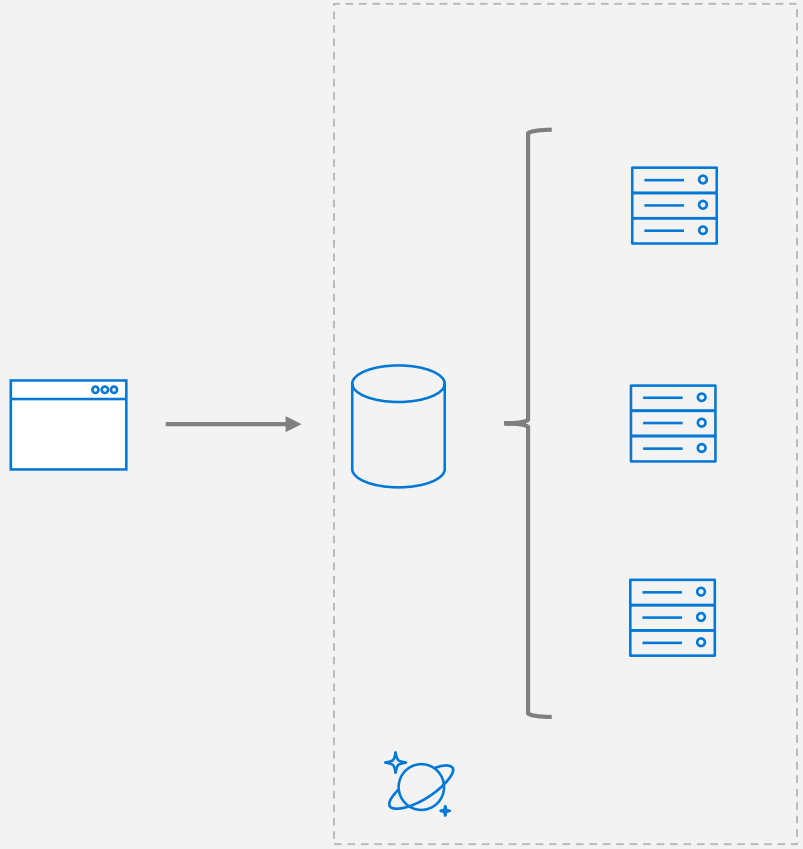
PARTITION DESIGN

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

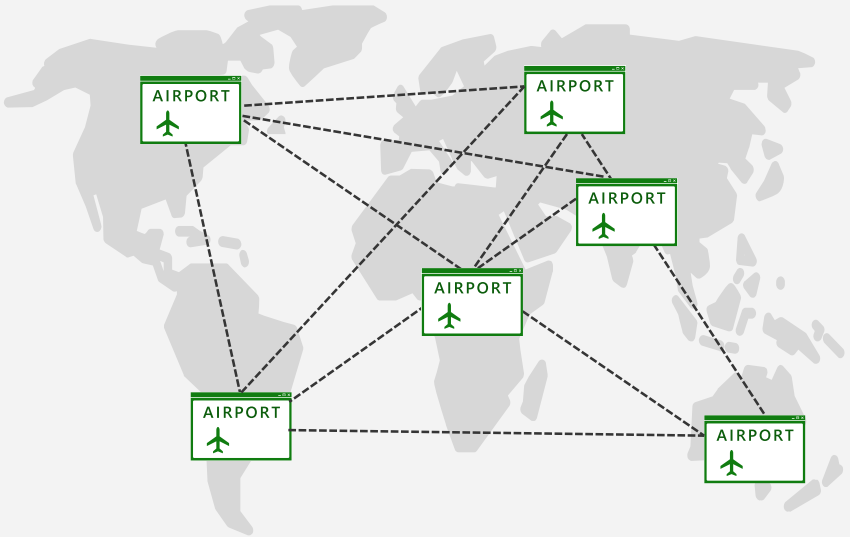
EXAMPLE SCENARIO

Contoso Air is an international airline serving customers around the world. They are planning to store their global flight data in Azure Cosmos DB.

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

WHAT ARE A FEW POTENTIAL PARTITION KEY CHOICES?

- Origin Airport
- Aircraft Type
- Date of Flight
- Operating Airline

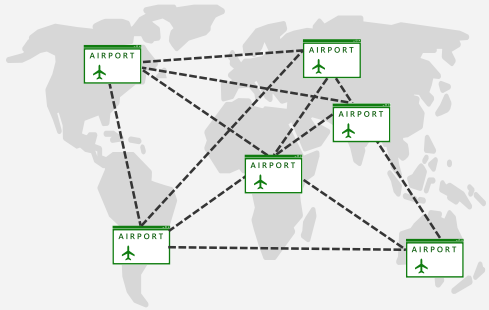


PARTITION KEY CHOICES

ORIGIN AIRPORT

Flights originate from many different airports. This number can change depending on whether an airline is regional, domestic or international.

You can also create a composite key that combines both the **origin** airport and the **destination** airport to increase granularity. (ex. SEA_JFK partition)



AIRCRAFT TYPE

This will create a typically fixed number of partitions. Most airlines only introduce a new aircraft type once every few years.

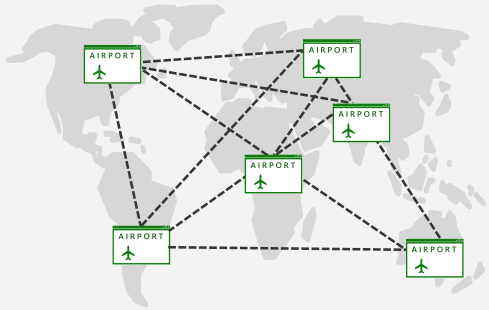
Some airlines rely on smaller aircraft for the majority of their flights and only use larger aircrafts for major routes.

PARTITION KEY CHOICES

DATE OF FLIGHT

Potentially the most granular option. If your workload is focused on queries by time (ex. I need a flight on Tuesday), this could be the ideal option.

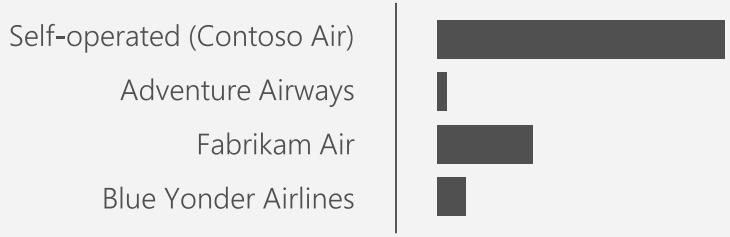
If you search across dates, this would require more cross-partition queries.



OPERATING AIRLINE

For the majority of airlines, partner-operated flights are a subset of total flights.

You could potentially create a single “hot” partition for the primary airline



PARTITION GRANULARITY

SELECT THE "RIGHT" LEVEL OF GRANULARITY FOR YOUR PARTITIONS

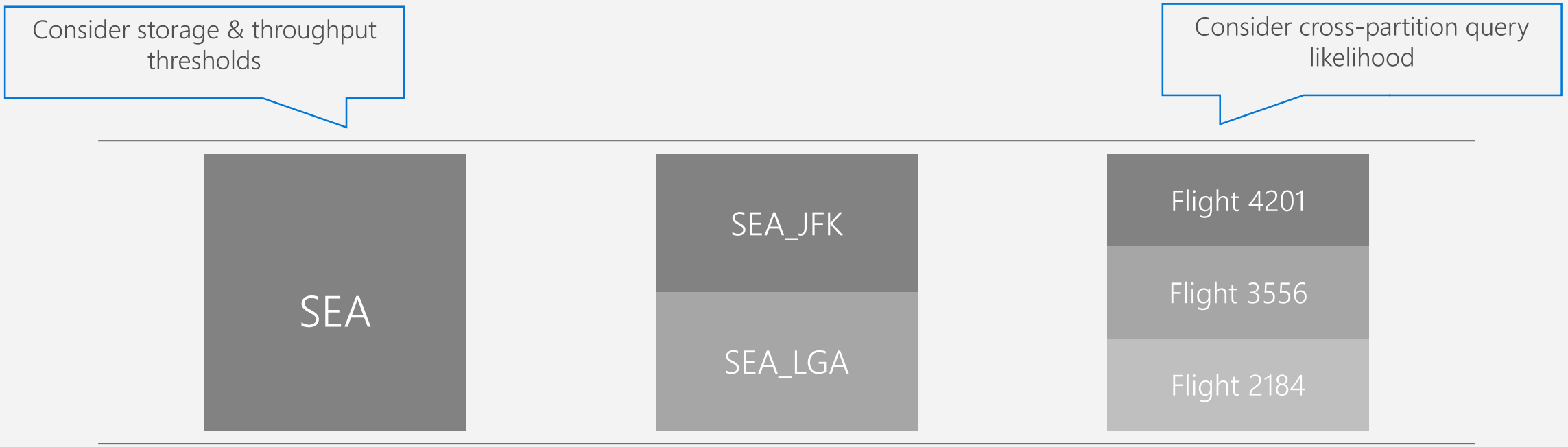
Partitions should be based on your most often occurring query and transactional needs. The goal is to **maximize granularity** and **minimize cross-partition requests**.

Don't be afraid to have more partitions!

More partition keys = More scalability

PARTITION GRANULARITY

SELECT THE "RIGHT" LEVEL OF GRANULARITY FOR YOUR PARTITIONS

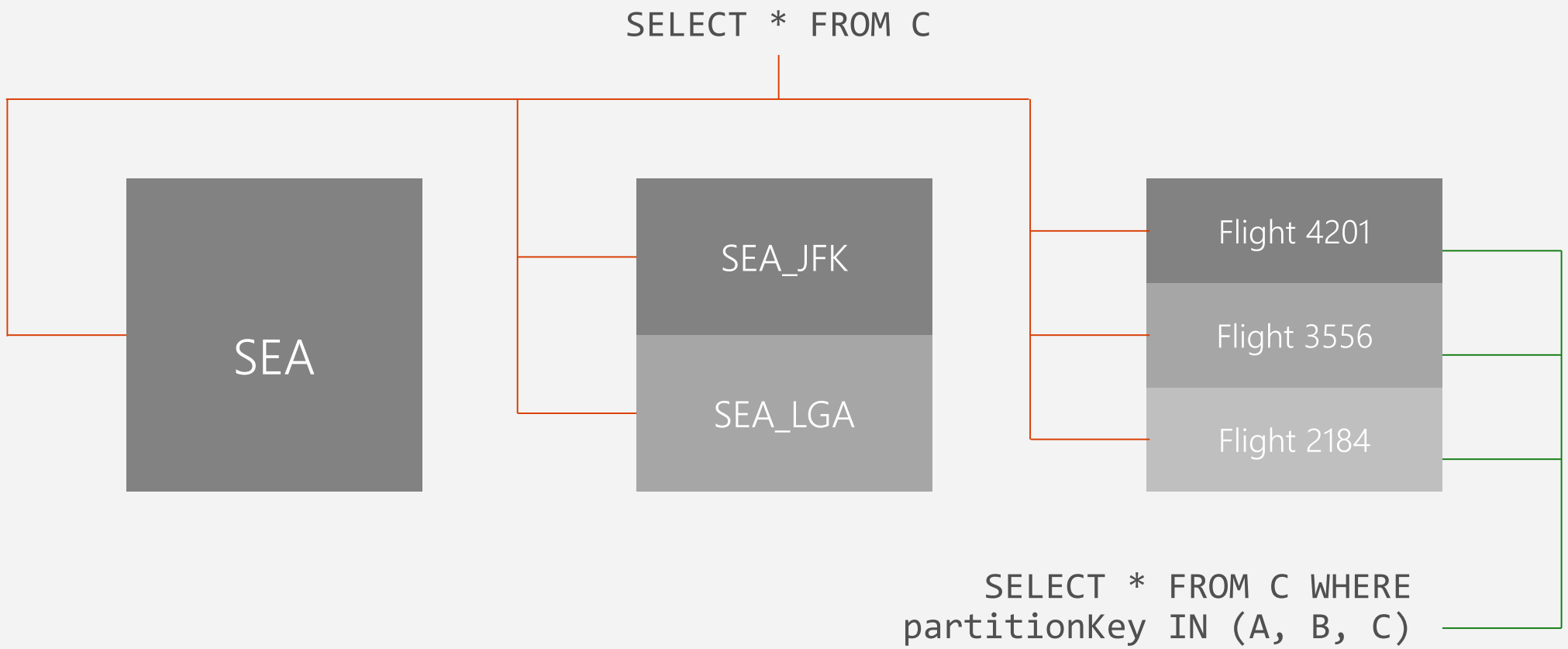


Don't be afraid to have more partitions!

More partition keys = More scalability

PARTITION GRANULARITY

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



PARTITIONS

Best Practices: Design Goals for Choosing a Good Partition Key

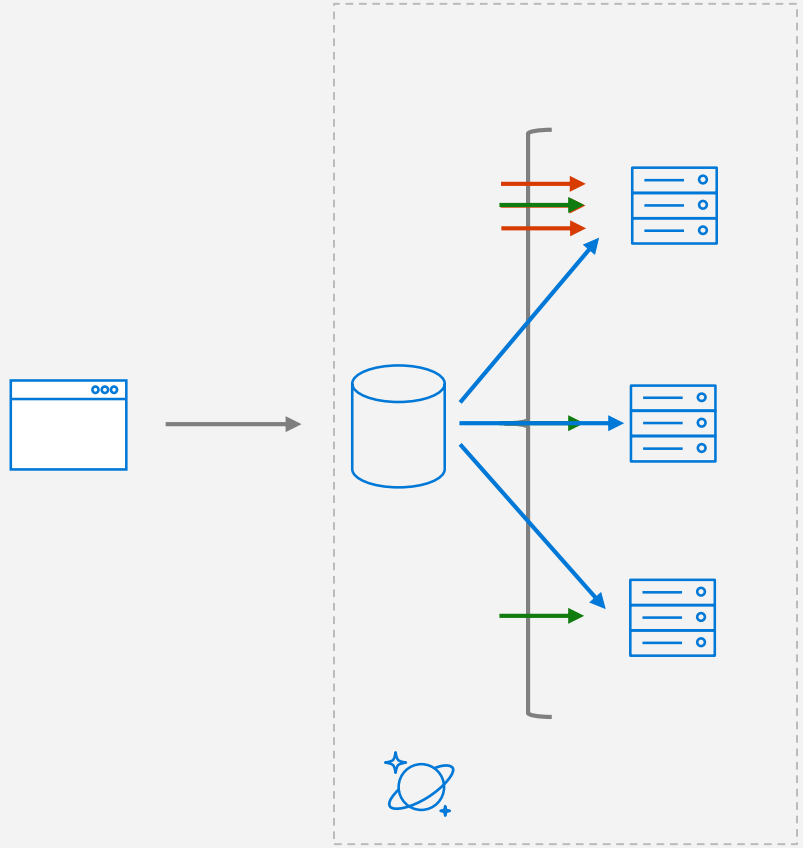
- Distribute the overall request + storage volume
  - Avoid "hot" partition keys
- Partition Key is scope for multi-record transactions and routing queries
  - Queries can be intelligently routed via partition key
  - Omitting partition key on query requires fan-out

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



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 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

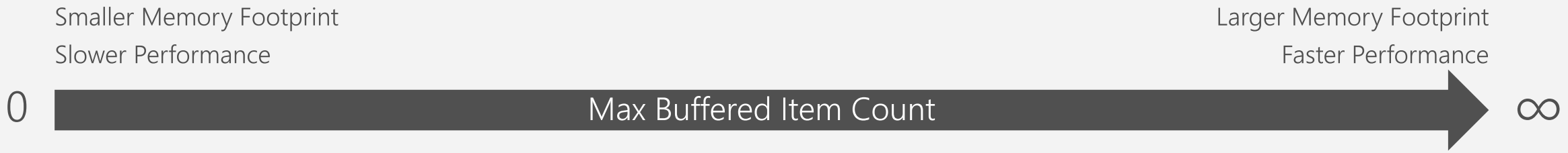
Use simple queries to enable greater scale.

Query execution metrics

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

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

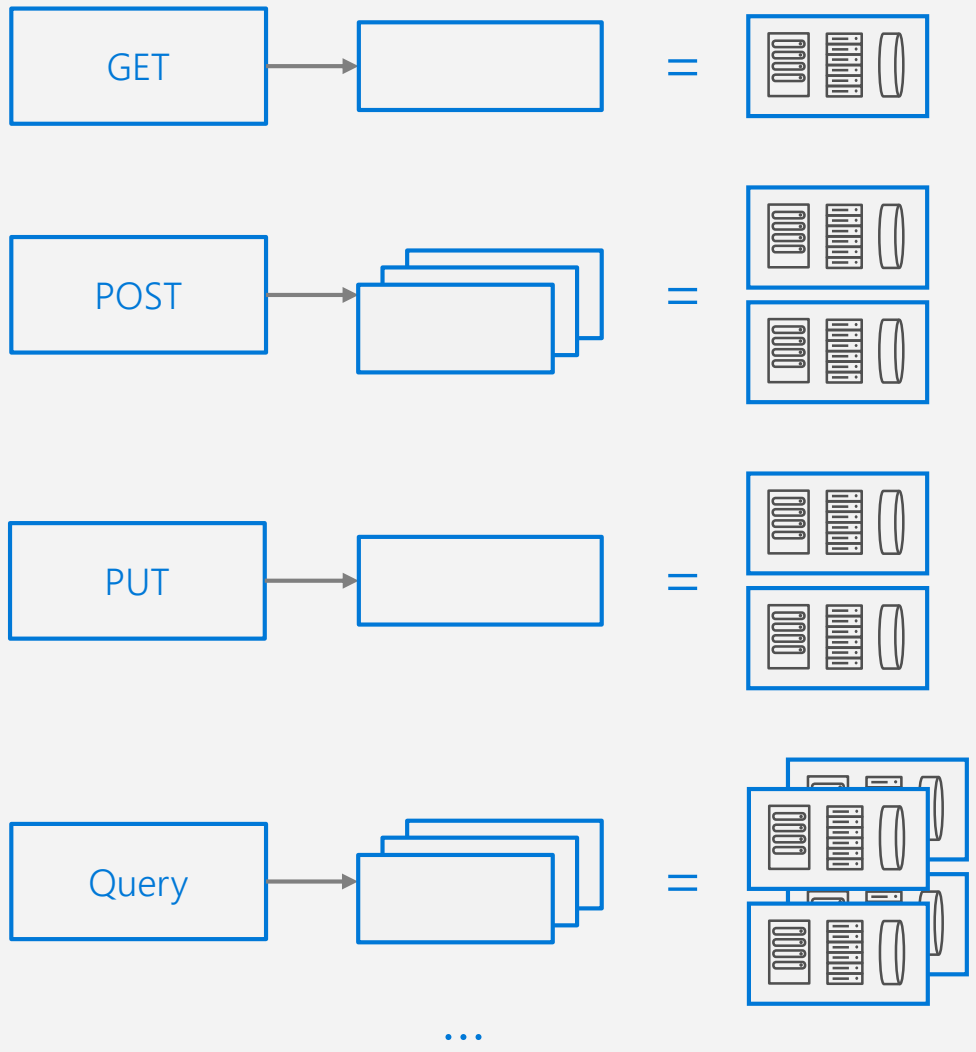
REQUEST UNITS

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

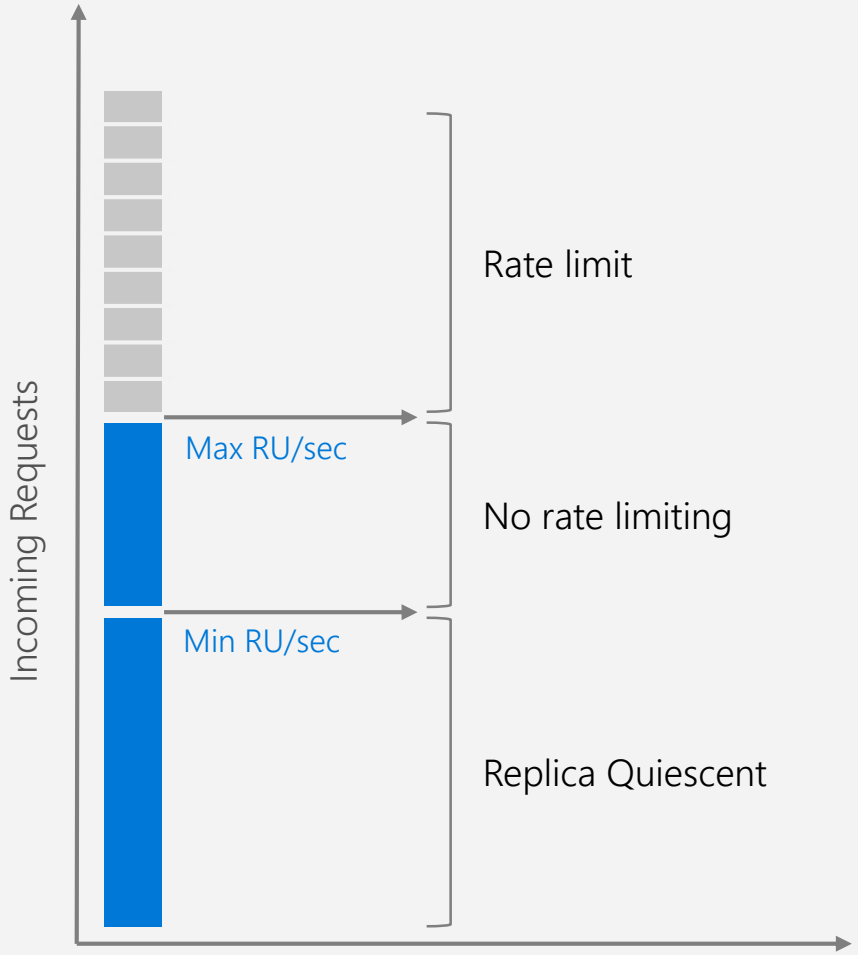
Rate limiting based on amount of throughput provisioned

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.

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

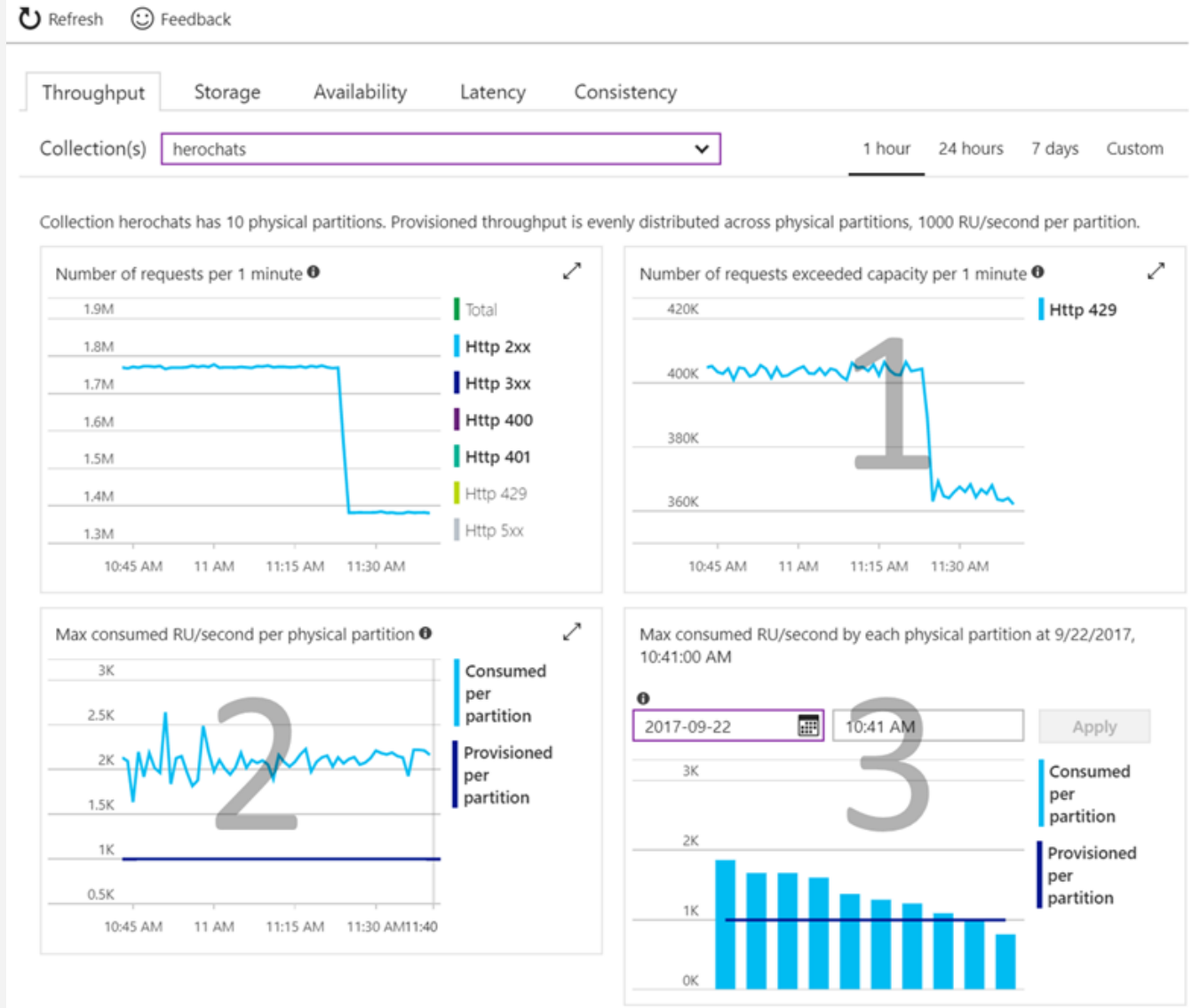
Total Monthly Cost

[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

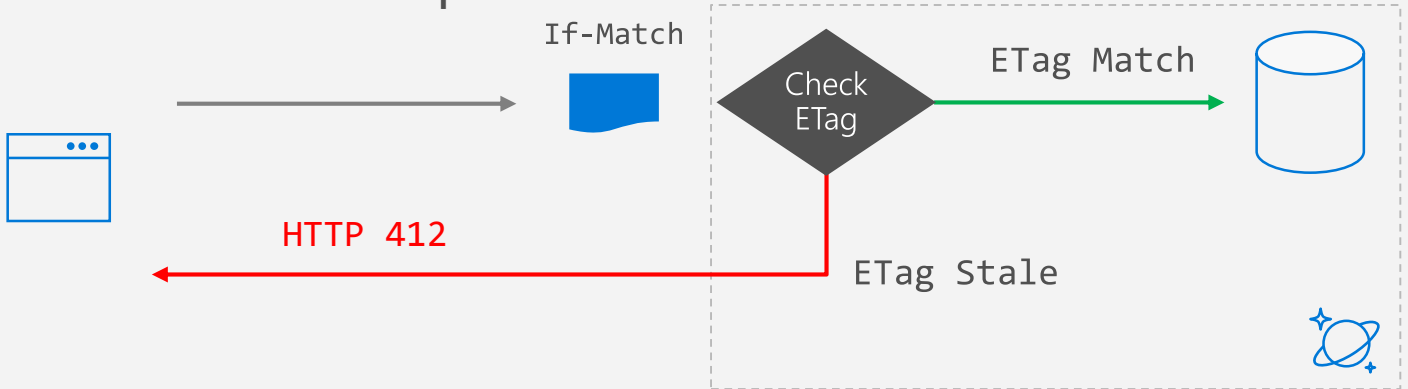
1. Check if your operations are getting rate limited.
    - Requests exceeding capacity chart
2. Check if consumed throughput exceeds the provisioned throughput on any of the physical partitions
    - Max RU/second consumed per partition chart
3. Select the time where the maximum consumed throughput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



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:**

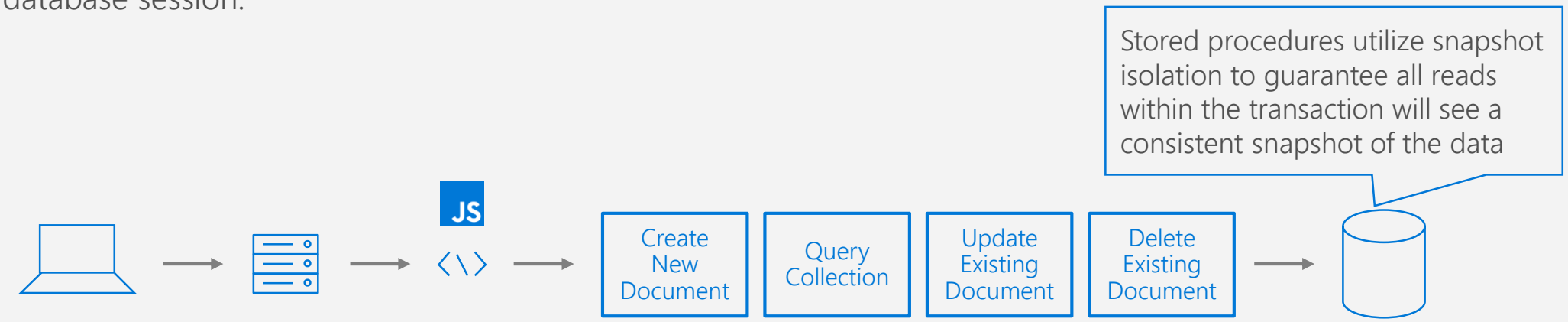


MULTI-DOCUMENT TRANSACTIONS

DATABASE TRANSACTIONS

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

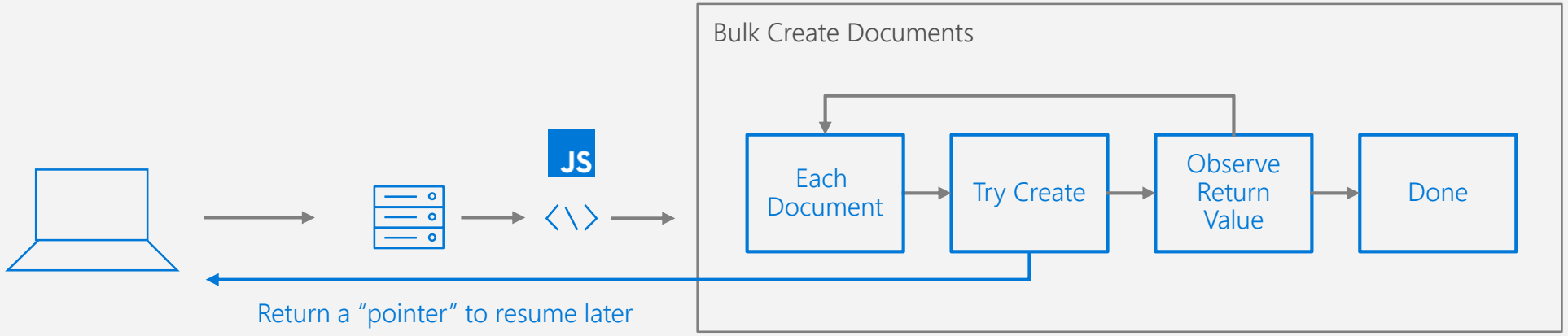
In Azure Cosmos DB, JavaScript is hosted in the same memory space as the database. Hence, requests made within stored procedures and triggers execute in the same scope of a database session.



TRANSACTION CONTINUATION MODEL

CONTINUING LONG-RUNNING TRANSACTIONS

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

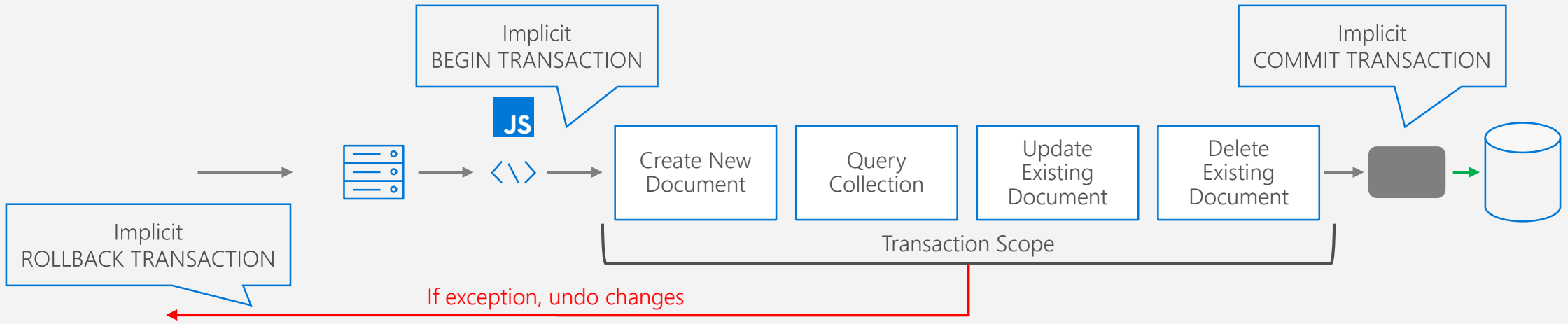


ROLLING BACK TRANSACTIONS

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.

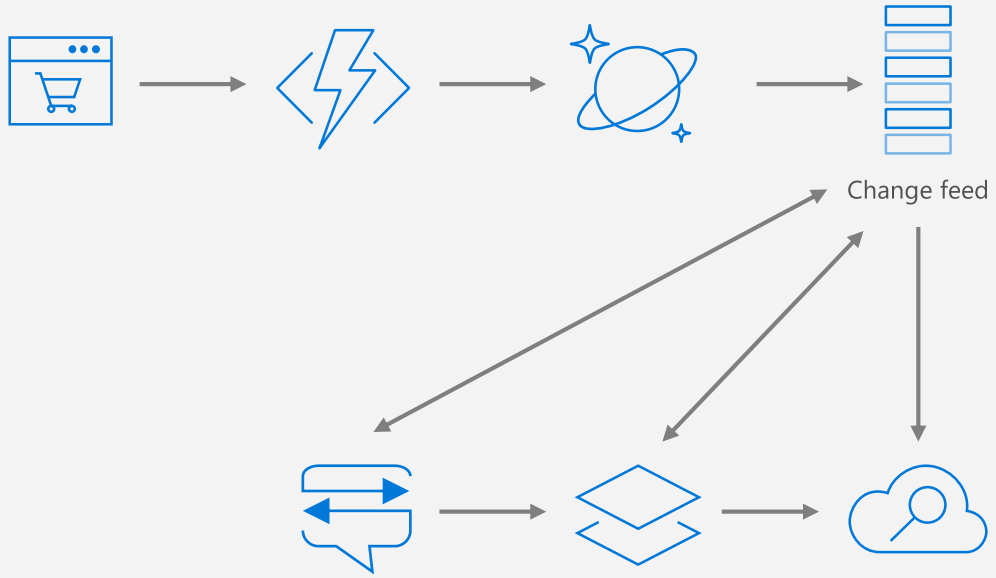


MODERN REACTIVE APPLICATIONS

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.

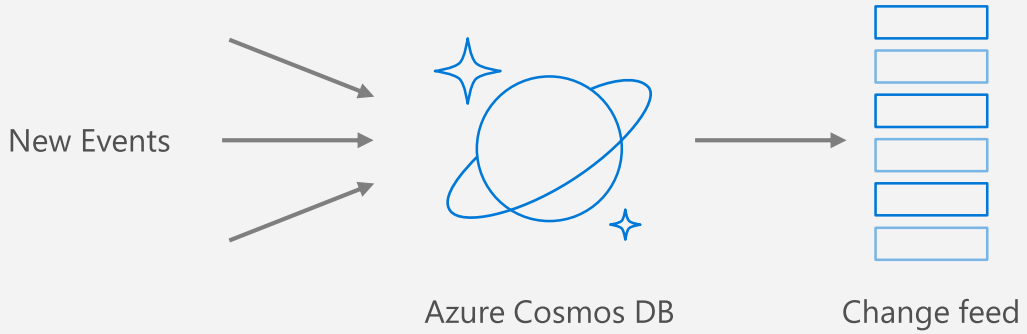
COMMON SCENARIOS

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

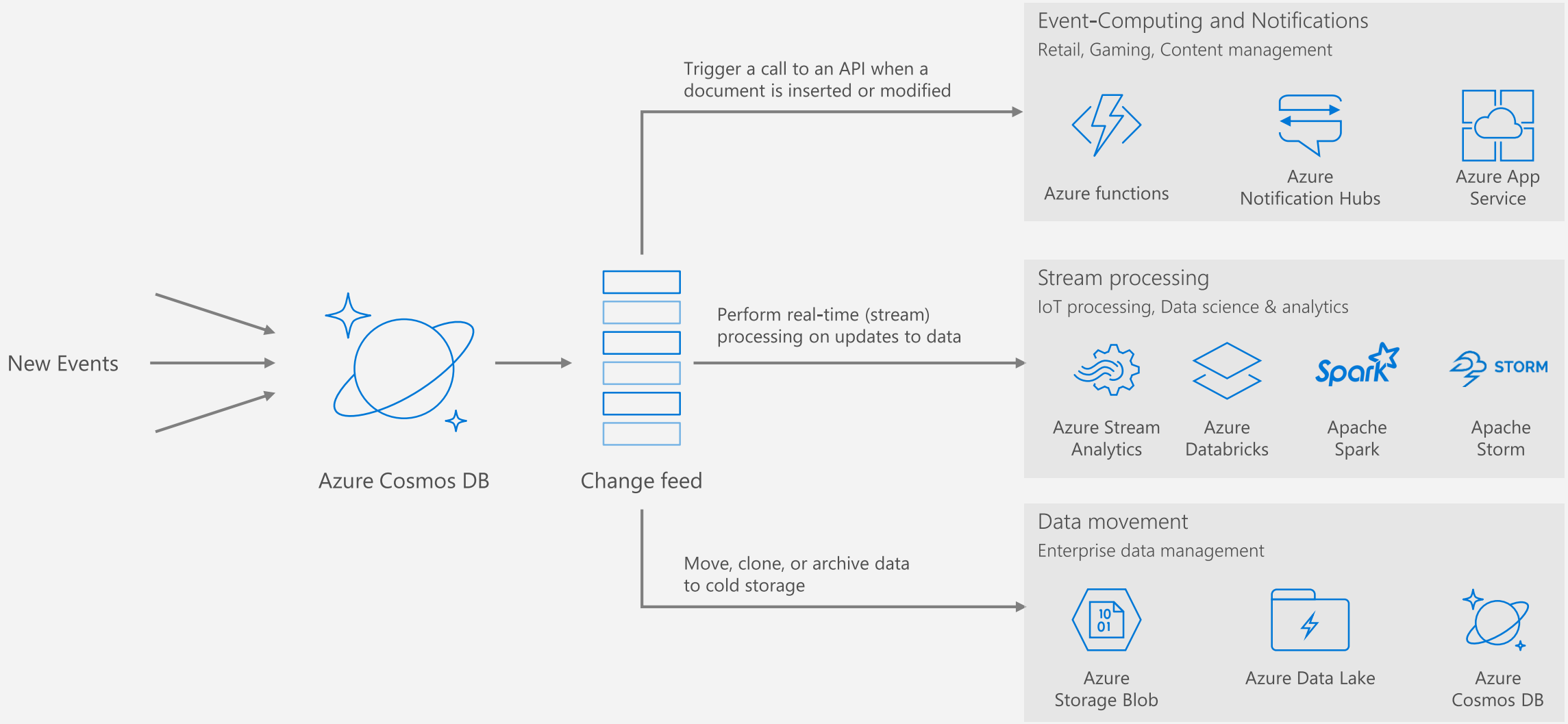


CHANGE FEED

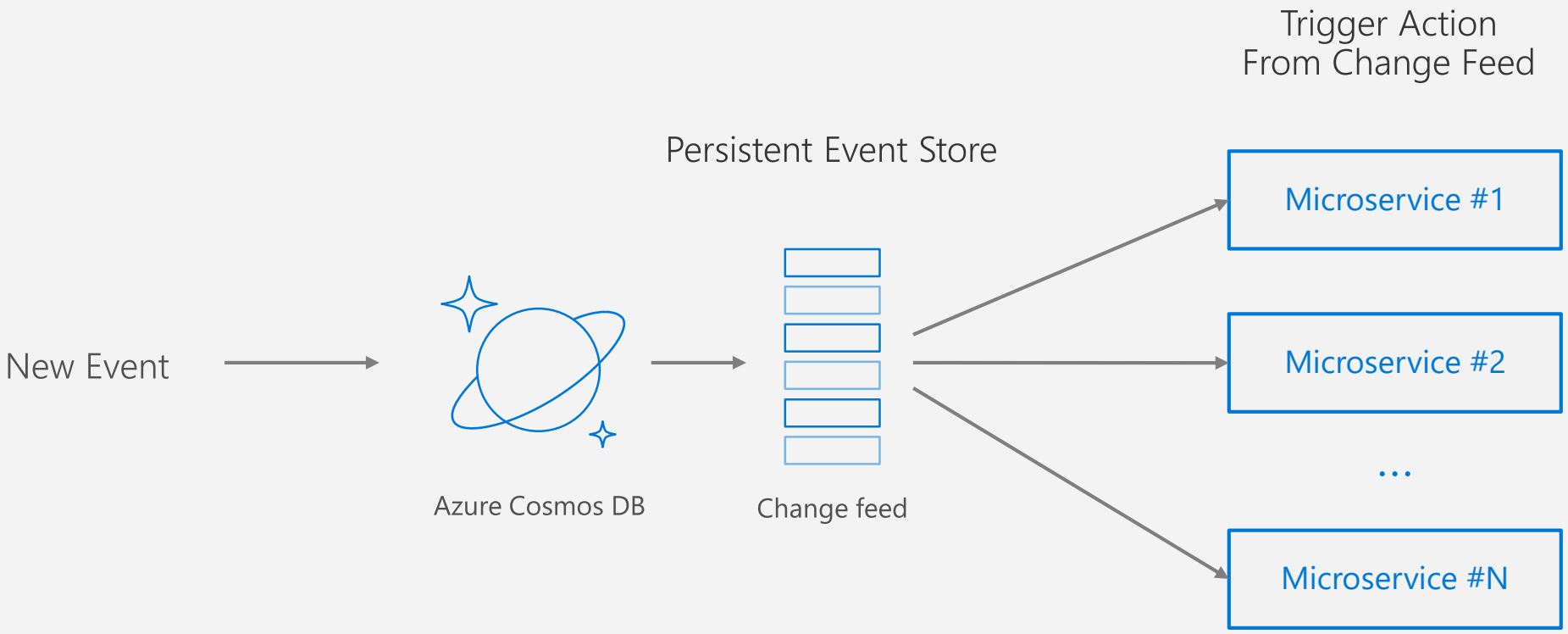
Persistent log of records within an Azure Cosmos DB container. Preseneted in the order in which they were modified



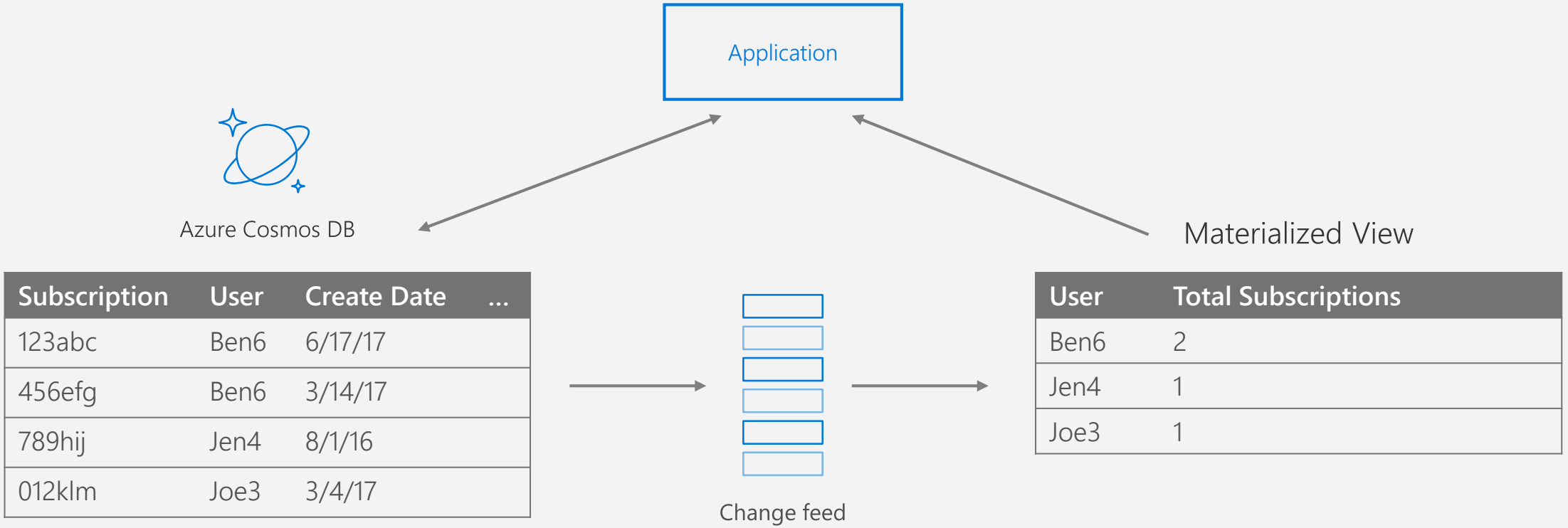
CHANGE FEED SCENARIOS



EVENT SOURCING FOR MICROSERVICES



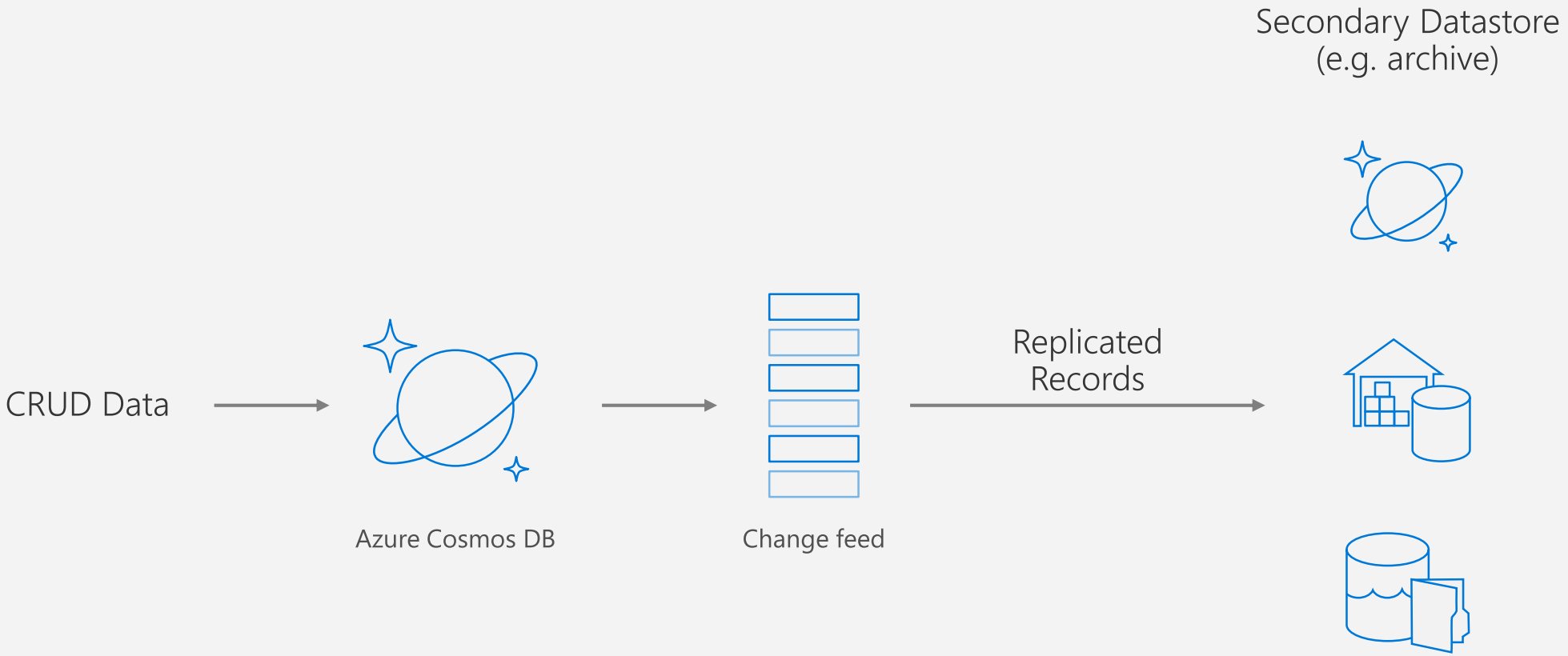
MATERIALIZING VIEWS



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

User	Total Subscriptions
Ben6	2
Jen4	1
Joe3	1

REPLICATING DATA



TIME-TO-LIVE (TTL)

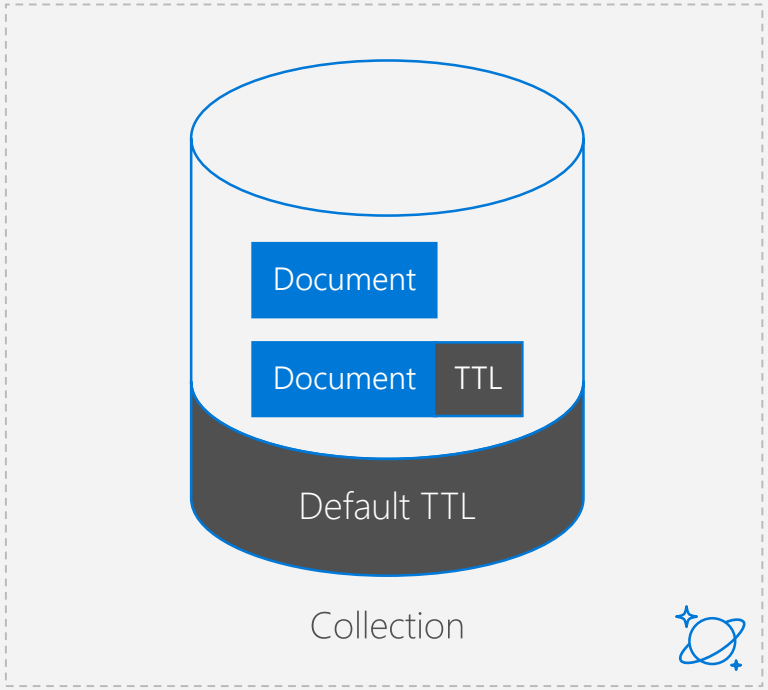
AUTOMATICALLY PURGE DATA

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.

- DefaultTTL for the collection
    - Missing/null - documents are not deleted
    - Value is "-1" - documents don't expire by default
    - Value is number "N" – documents expire N seconds after last modification
- TTL for the documents:
    - Overrides the DefaultTTL value for the parent collection.



© 2018 Microsoft Corporation. All rights reserved.