



# Azure Cosmos DB

Technical Deep Dive

Jorge Borralho

Technical Lead @ Digital Xperience Xpand IT

[jorge.borralho@xpand-it.com](mailto:jorge.borralho@xpand-it.com)

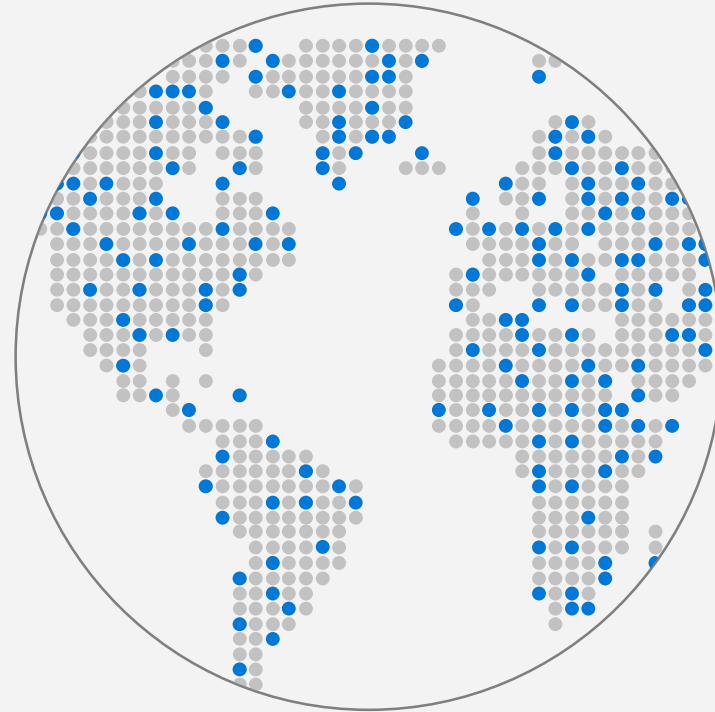


# TURNKEY GLOBAL DISTRIBUTION

## 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

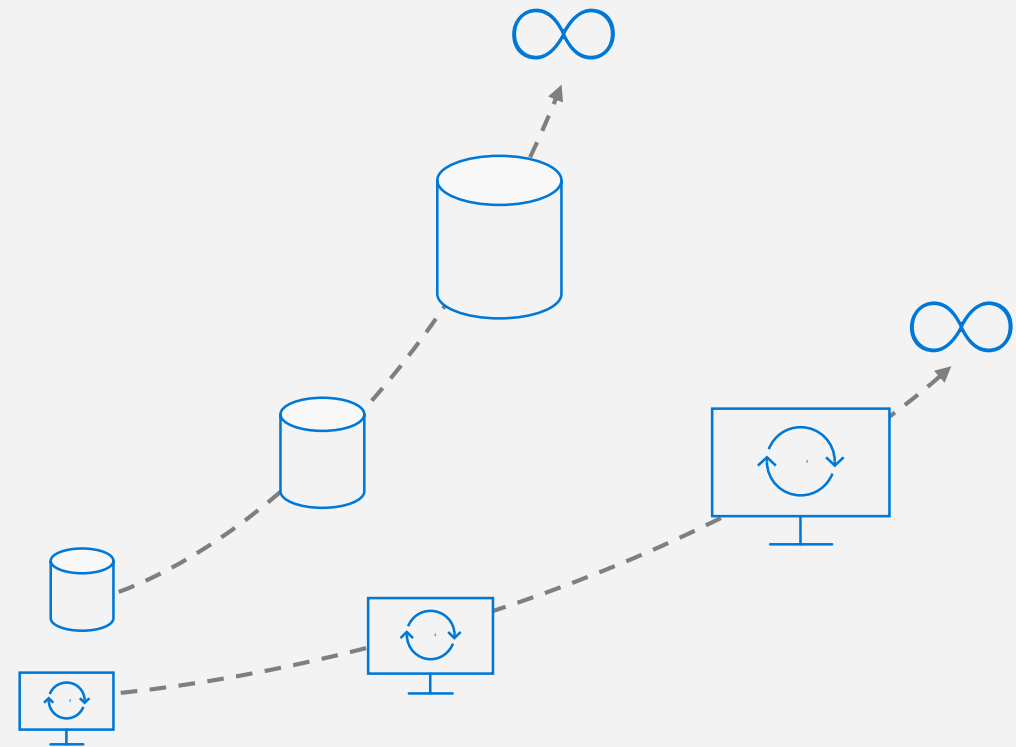


# ELASTIC SCALE OUT OF STORAGE AND THROUGHPUT

## 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.

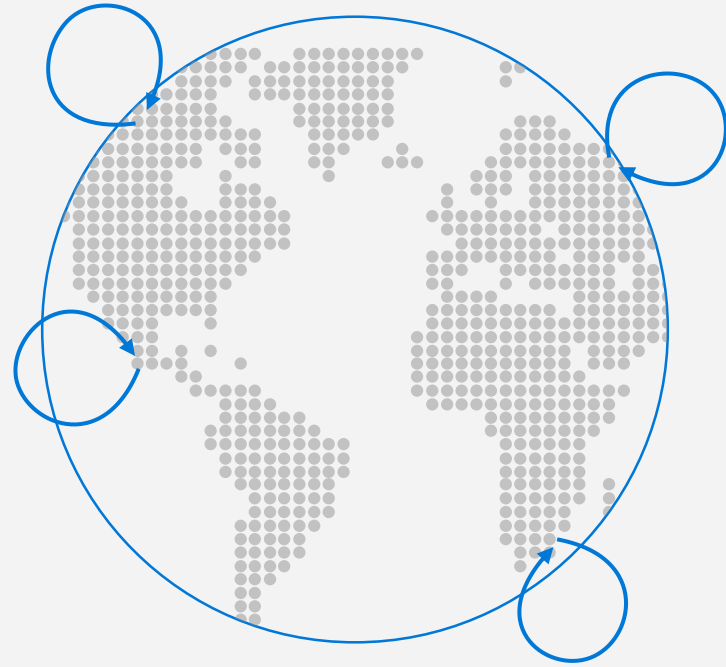
- Elastically scale throughput from 10 to 100s of millions of requests/sec across multiple regions
- Support for requests/sec for different workloads
- Pay only for the throughput and storage you need



# 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.



Strong



Bounded-stateless



Session



Consistent prefix



Eventual



# MULTIPLE DATA MODELS AND APIS

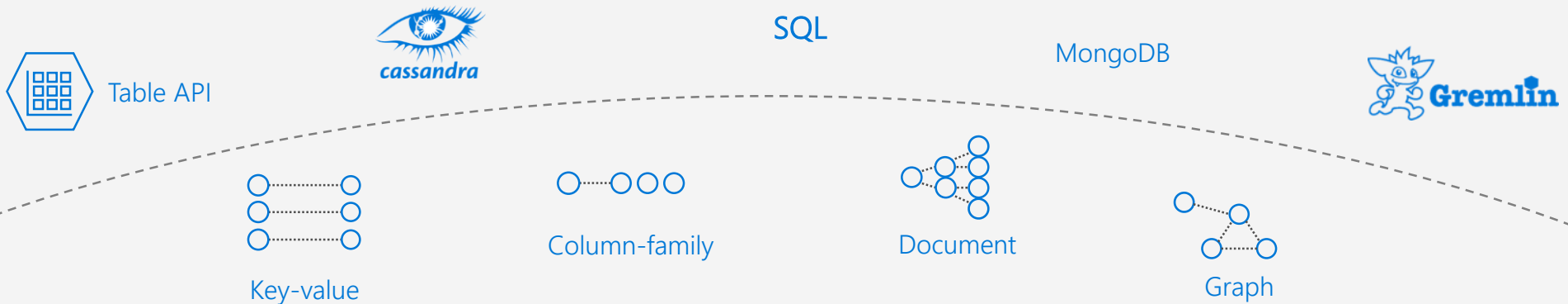
## USE THE MODEL THAT FITS YOUR REQUIREMENTS, AND THE APIS, TOOLS, AND FRAMEWORKS YOU PREFER

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



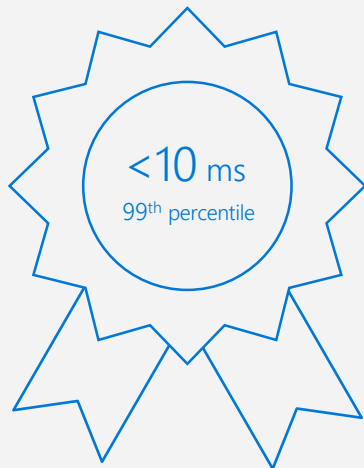


# COMPREHENSIVE SLAs

## 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



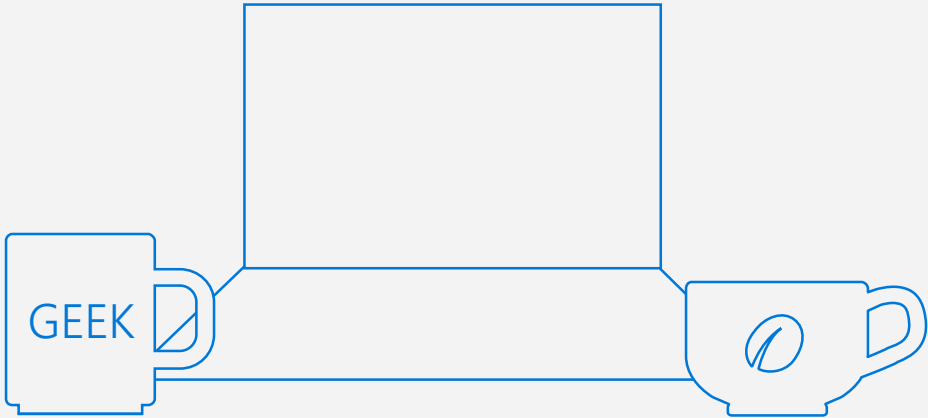
### Consistency



# 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



# Use Cases

# TOP 10 REASONS WHY CUSTOMERS USE AZURE COSMOS DB



The 1<sup>st</sup> and only database with **global distribution** turnkey capability



Deliver **massive storage/throughput scalability** database



Provides guaranteed **single digit millisecond latency** at 99<sup>th</sup> 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**

# 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



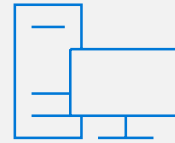
**Build real-time customer experiences**

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



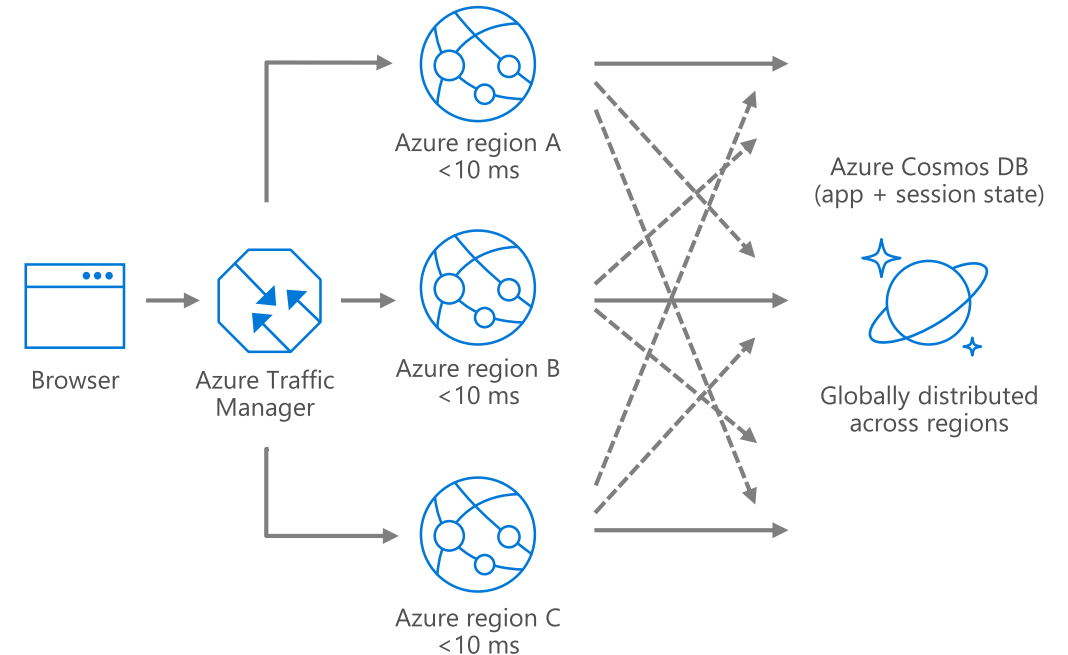
**Lift and shift NoSQL data**

Lift and shift MongoDB and Cassandra workloads

# 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.

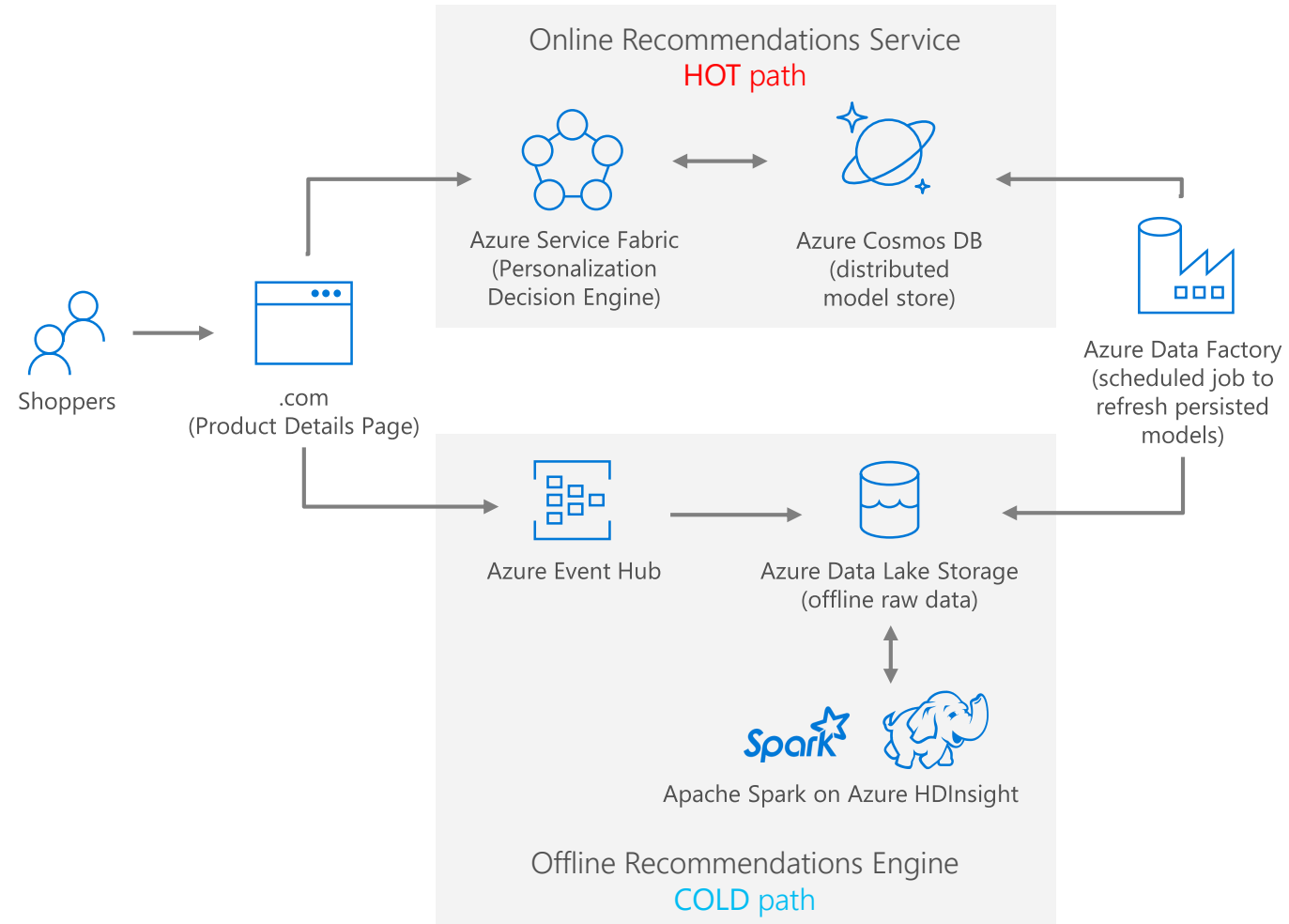
- Turnkey global data replication across all Azure regions
- Guaranteed low-latency experience for global users
- Resiliency for high availability and disaster recovery



# BUILD REAL-TIME CUSTOMER EXPERIENCES

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

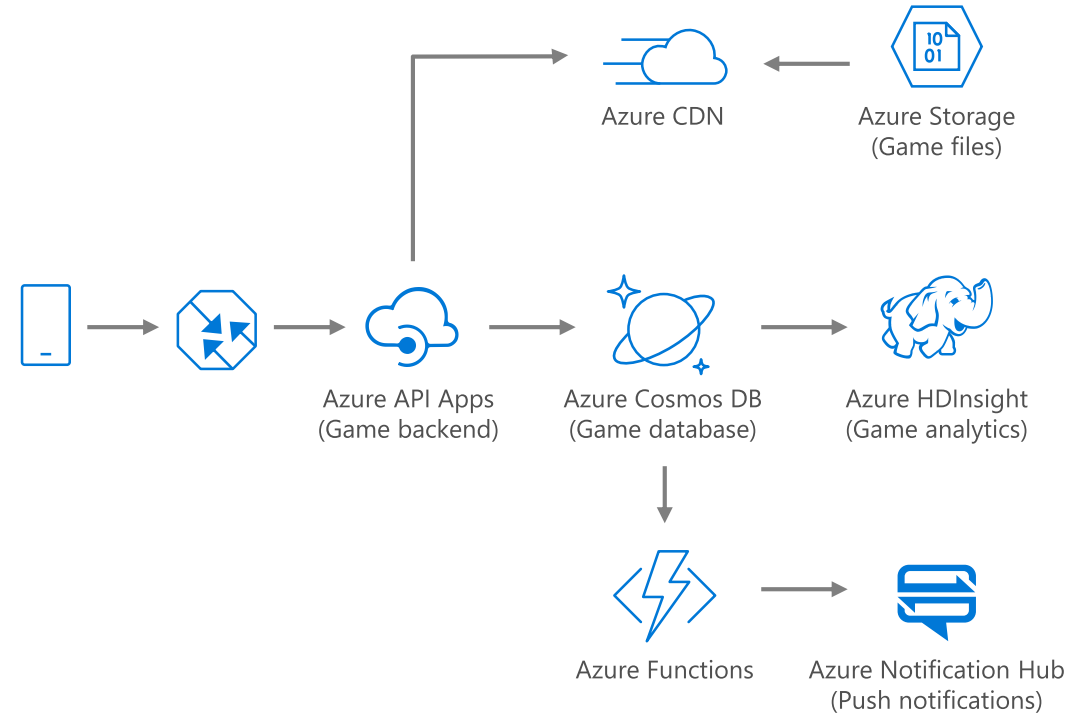
- Machine learning models generate real-time recommendations across product catalogues
- Product analysis in milliseconds
- Low-latency ensures high app performance worldwide
- Tunable consistency models for rapid insight



# IDEAL FOR GAMING 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

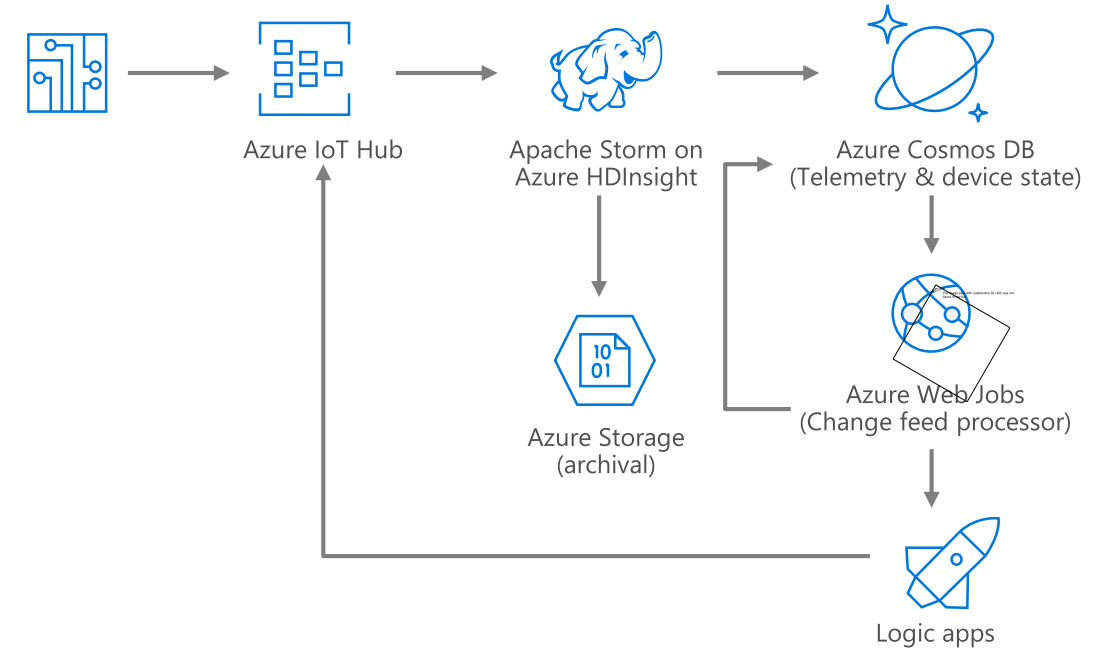




# MASSIVE SCALE TELEMETRY STORES FOR IOT

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



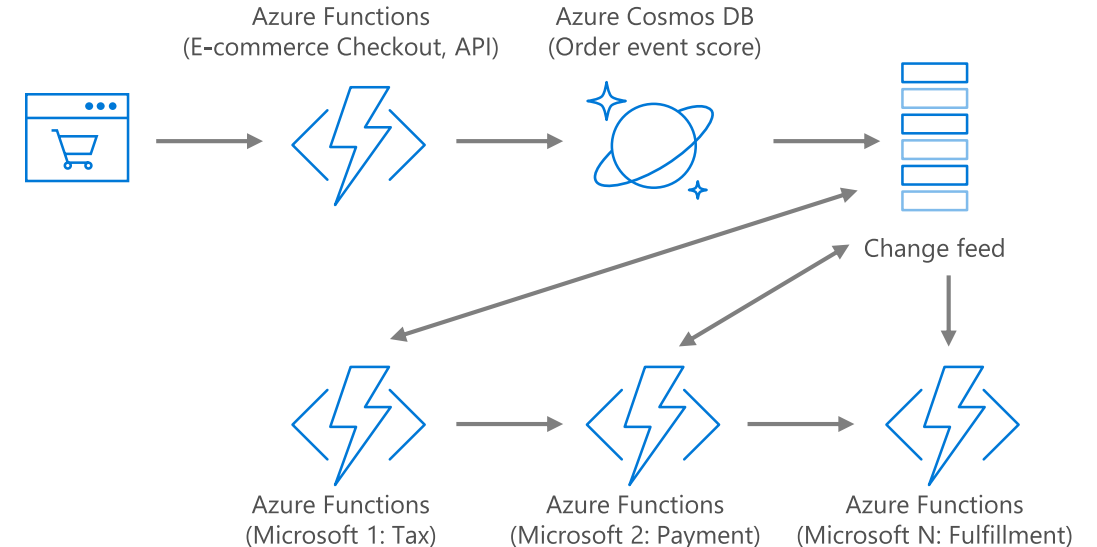
**Honeywell**



# SIMPLIFIED DEVELOPMENT WITH SERVERLESS ARCHITECTURE

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

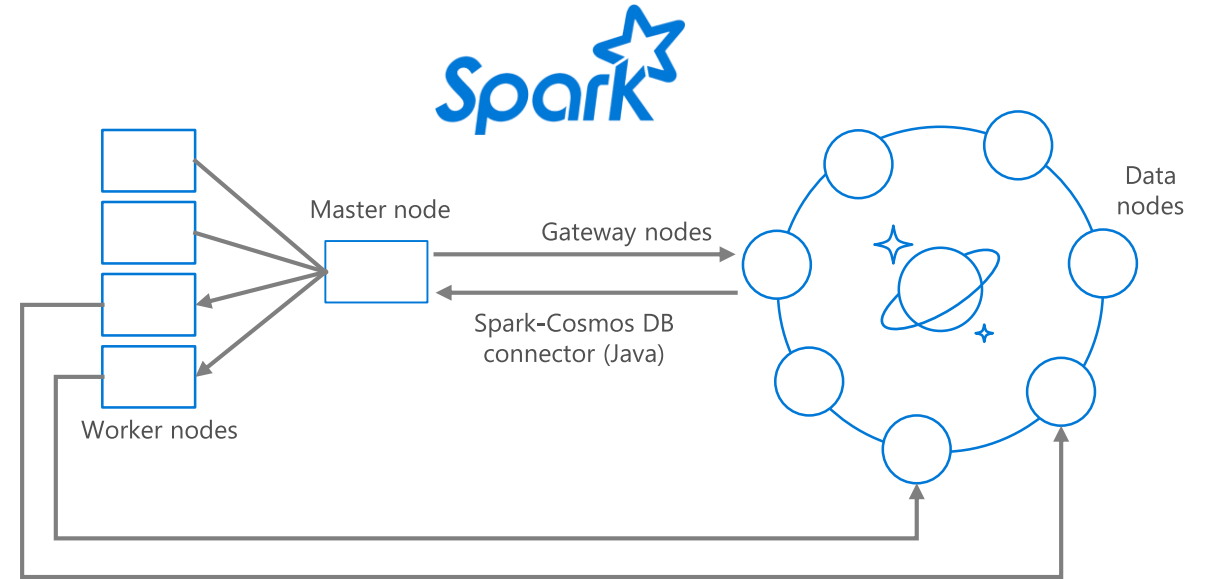
- Seamless handling of any data output or volume
- Data made available immediately, and indexed automatically
- High writes per second, with stable ingestion and query performance
- Real-time, resilient change feeds logged forever and always accessible
- Native integration with Azure Functions



# RUN SPARK OVER OPERATIONAL DATA

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

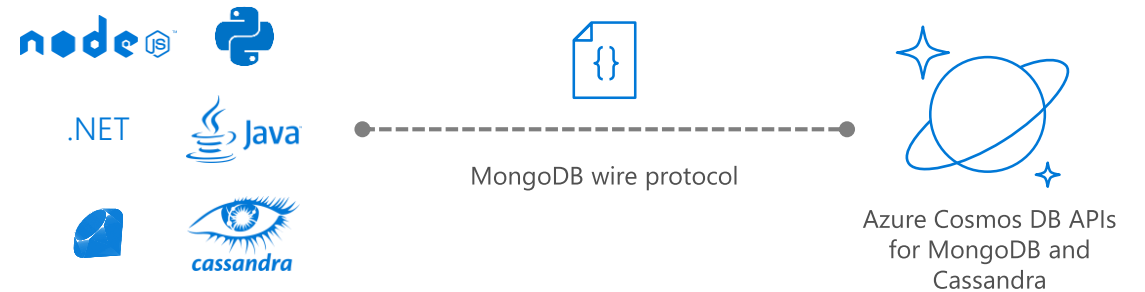
- Real-time big data processing across any data model
- Machine learning at scale over globally-distributed data
- Speeds analytical queries with automatic indexing and push-down predicate filtering
- Native integration with Spark Connector



# 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



# Data Modelling

# Data modelling

- Is just as important with relational data!
- There's still a schema – just enforced at the application level
- Plan upfront for best performance & costs
- Answer: Smart data modelling will help

# Modelling challenges

- #1: To de-normalize, or normalize? To embed, or to reference?
- #2: Put data types in same collection, or different?




# Modeling challenge #1: To embed or reference?

## Embed

```
{
  "menuID": 1,
  "menuName": "Lunch menu",
  "items": [
    {"ID": 1, "ItemName": "hamburger", "ItemDescription": "..."}
    {"ID": 2, "ItemName": "cheeseburger", "ItemDescription": "..."}
  ]
}
```

## Reference

```
{
  "menuID": 1,
  "menuName": "Lunch menu",
  "items": [
    {"ID": 1}
    {"ID": 2}
  ]
}
```



The diagram shows an arrow pointing from the `{"ID": 1}` object in the `items` array of the `Reference` JSON to the first object in the `items` array of the `Embed` JSON, illustrating that the `Reference` version uses a reference to a separate object rather than embedding the full object.

```
{
  "ID": 1, "ItemName": "hamburger", "ItemDescription": "..."}
{"ID": 2, "ItemName": "cheeseburger", "ItemDescription": "..."}
}
```

# When to embed #1

- “Data that is queried together, should live together”

```
{  
  "ID": 1,  
  "ItemName": "hamburger",  
  "ItemDescription": "cheeseburger, no cheese",  
  "Category": "sandwiches",  
  "CategoryDescription": "2 pieces of bread + filling",  
  "Ingredients": [  
    {"ItemName": "bread", "calorieCount": 100, "Qty": "2 slices"},  
    {"ItemName": "lettuce", "calorieCount": 10, "Qty": "1 slice"},  
    {"ItemName": "tomato", "calorieCount": 10, "Qty": "1 slice"},  
    {"ItemName": "patty", "calorieCount": 700, "Qty": "1"}  
  ]  
}
```

E.g. in Recipe, ingredients are always queried with the item

## When to embed #2

- Child data is dependent/intrinsic to a parent

```
{  
  "id": "Order1",  
  "customer": "Customer1",  
  "orderDate": "2018-09-26",  
  "itemsOrdered": [  
    {"ID": 1, "ItemName": "hamburger", "Price":9.50,  
     "Qty": 1}  
    {"ID": 2, "ItemName": "cheeseburger", "Price":9.50,  
     "Qty": 499}  
  ]  
}
```

Items Ordered depends on Order

# When to embed #3

- 1:1 relationship

```
{  
  "id": "1",  
  "name": "Alice",  
  "email": "alice@contoso.com",  
  "phone": "555-5555",  
  "loyaltyNumber": 13838359,  
  "addresses": [  
    {"street": "1 Contoso Way", "city": "Seattle"},  
    {"street": "15 Fabrikam Lane", "city":  
      "Orlando"}  
  ]  
}
```

All customers have email, phone, loyalty number for 1:1 relationship

# When to embed #4, #5

- Similar rate of updates – does the data change at the same (slower) pace? -> Minimize writes
- 1:few relationships

```
{
  "id": "1",
  "name": "Alice",
  "email": "alice@contoso.com",      //Email, addresses don't change too often
  "addresses": [
    {"street": "1 Contoso Way", "city":
    "Seattle"},
    {"street": "15 Fabrikam Lane", "city":
    "Orlando"}
  ]
}
```

# When to embed - summary

- Data from entities is queried together
  - Child data is dependent on a parent
  - 1:1 relationship
  - Similar rate of updates – does the data change at the same pace
  - 1:few – the set of values is bounded
- 
- Usually embedding provides better read performance
  - Follow-above to minimize trade-off for write perf

# When to reference #1

1 : many (unbounded relationship)

```
{
  "id": "1",
  "name": "Alice",
  "email": "alice@contoso.com",
  "Orders": [
    {
      "id": "Order1",
      "orderDate": "2018-09-18",
      "itemsOrdered": [
        {"ID": 1, "ItemName": "hamburger", "Price": 9.50, "Qty": 1},
        {"ID": 2, "ItemName": "cheeseburger", "Price": 9.50, "Qty": 499}
      ],
    },
    ...
    {
      "id": "OrderNfinity",
      "orderDate": "2018-09-20",
      "itemsOrdered": [
        {"ID": 1, "ItemName": "hamburger", "Price": 9.50, "Qty": 1}
      ],
    }
  ]
}
```

**Embedding doesn't make sense:**

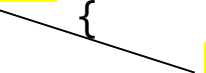
- **Too many writes to same document**
- **2MB document limit**



# When to reference #1

1 : many (unbounded relationship)

```
{
  "id": "1",
  "name": "Alice",
  "email": "alice@contoso.com",
  "Orders": ["Order1", .. "Order100"]
}
{
  "id": "Order1",
  "orderDate": "2018-09-18",
  "itemsOrdered": [
    {"ID": 2, "Name": "cheeseburger", "Price": 9.50, "Qty": 5}
  ]
},
{
  "id": "Order100",
  "orderDate": "2018-09-20",
  "itemsOrdered": [
    {"ID": 1, "Name": "hamburger", "Price": 9.50, "Qty": 1}
  ]
},
```

A line connects the "Order100" element in the "Orders" array of the first object to the "id" field of the second object in the array, illustrating a reference.

# When to reference #2

## Data changes at different rates #2

```
{  
  "id": "1",  
  "name": "Alice",  
  "email": "alice@contoso.com",  
  "stats": [  
    {"TotalNumberOrders": 100},  
    {"TotalAmountSpent": 550}]  
}
```

Number of orders, amount spent will likely change faster than email

**Guidance: Store these aggregate data in own document, and reference it**

# When to reference #3

many : many relationships

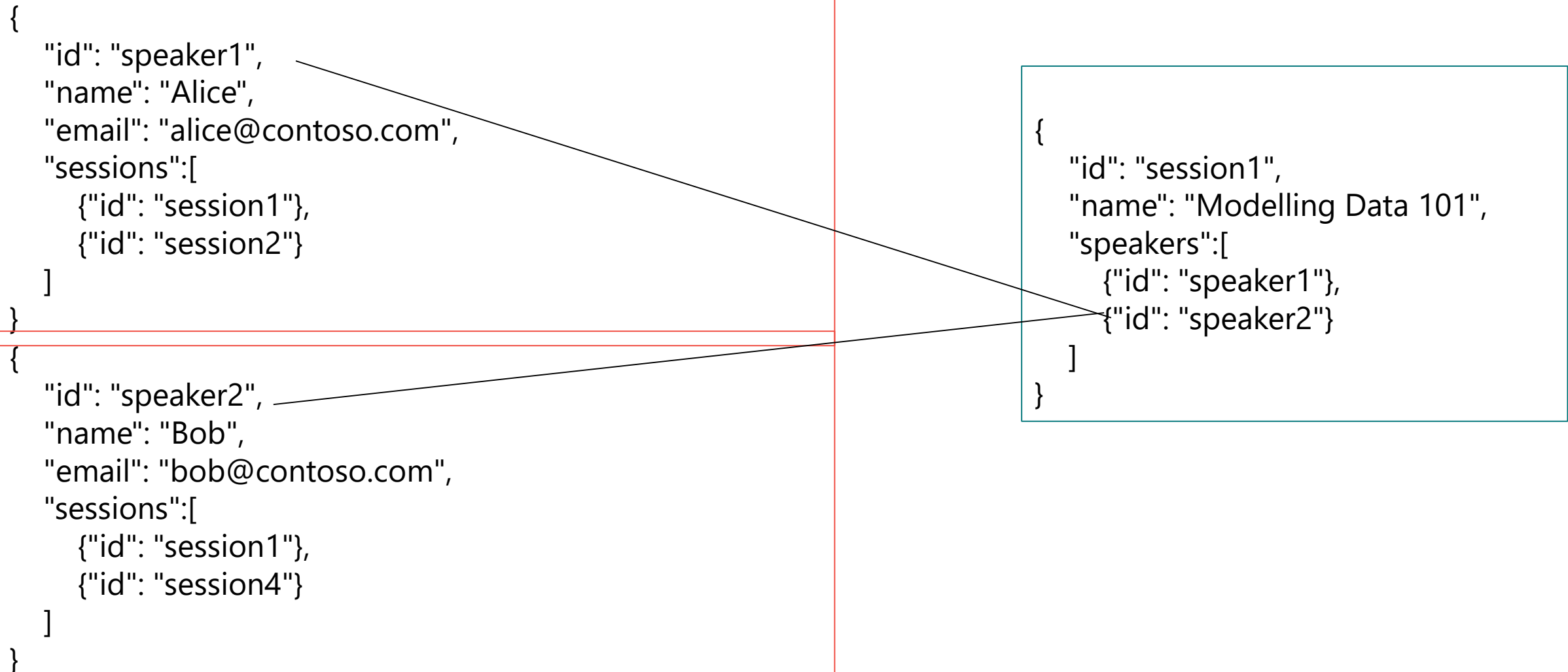
Speakers have multiple sessions  
Sessions have multiple speakers

Have Speaker & Session documents

```
{
  "id": "speaker1",
  "name": "Alice",
  "email": "alice@contoso.com",
  "sessions": [
    {"id": "session1"},
    {"id": "session2"}
  ]
}
```

```
{
  "id": "speaker2",
  "name": "Bob",
  "email": "bob@contoso.com",
  "sessions": [
    {"id": "session1"},
    {"id": "session4"}
  ]
}
```

```
{
  "id": "session1",
  "name": "Modelling Data 101",
  "speakers": [
    {"id": "speaker1"},
    {"id": "speaker2"}
  ]
}
```



# When to reference #4

What is referenced, is heavily referenced by many others

```
{
  "id": "speaker1",
  "name": "Alice",
  "email": "alice@contoso.com",
  "sessions": [
    {"id": "session1"},
    {"id": "session2"}
  ]
}
```

```
{
  "id": "attendee1",
  "name": "Eve",
  "email": "eve@contoso.com",
  "bookmarkedSessions": [
    {"id": "session1"},
    {"id": "session4"}
  ]
}
```

```
{
  "id": "session1",
  "name": "Modelling Data 101",
  "speakers": [
    {"id": "speaker1"},
    {"id": "speaker2"}
  ]
}
```

Here, session is referenced by speakers and attendees

Allows you to update Session independently

# When to reference summary

1 : many (unbounded relationship)

many : many relationships

Data changes at different rates

What is referenced, is heavily referenced by many others

Typically provides better write performance

But may require more network calls for reads

# But wait, you can do both!

```
{
  "id": "speaker1",
  "name": "Alice",
  "email": "alice@contoso.com",
  "address": "1 Microsoft Way",
  "phone": "555-5555",
  "sessions": [
    {"id": "session1"},
    {"id": "session2"}
  ]
}
```

Speaker

```
{
  "id": "session1",
  "name": "Modelling Data 101",
  "speakers": [
    {"id": "speaker1", "name": "Alice", "email": "alice@contoso.com"},
    {"id": "speaker2", "name": "Bob"}
  ]
}
```

Session

**Embed** frequently used data, but use the **reference** to get **less frequently used**

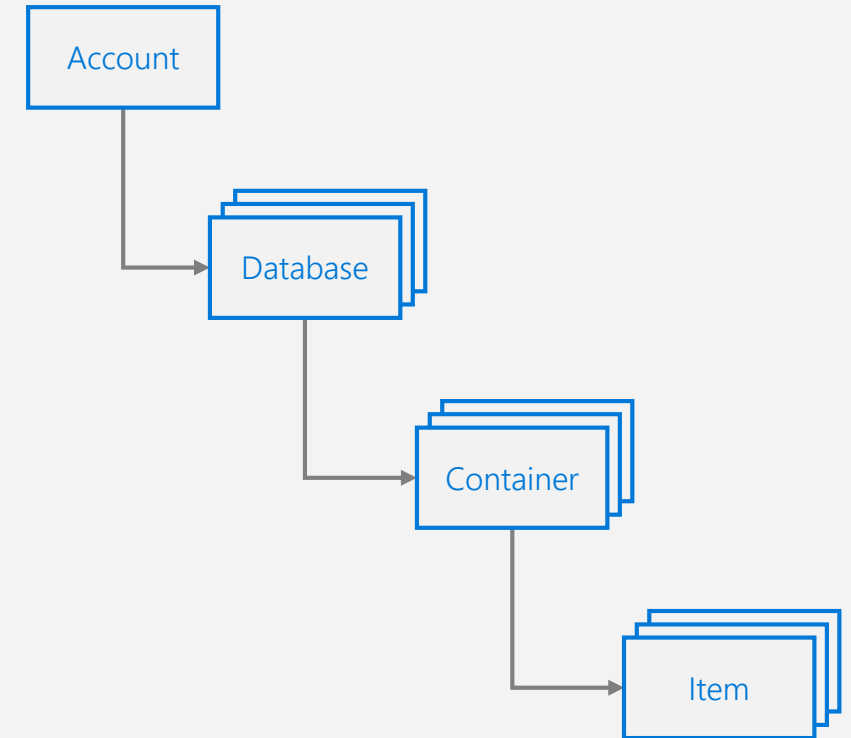
# Partitioning



# RESOURCE MODEL

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

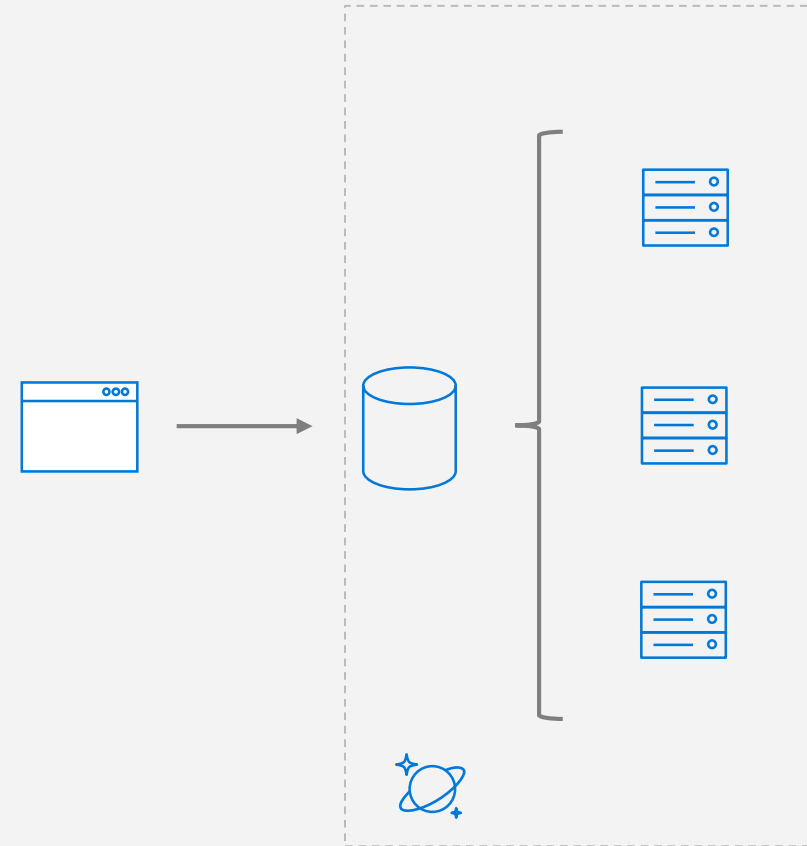
*This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.*



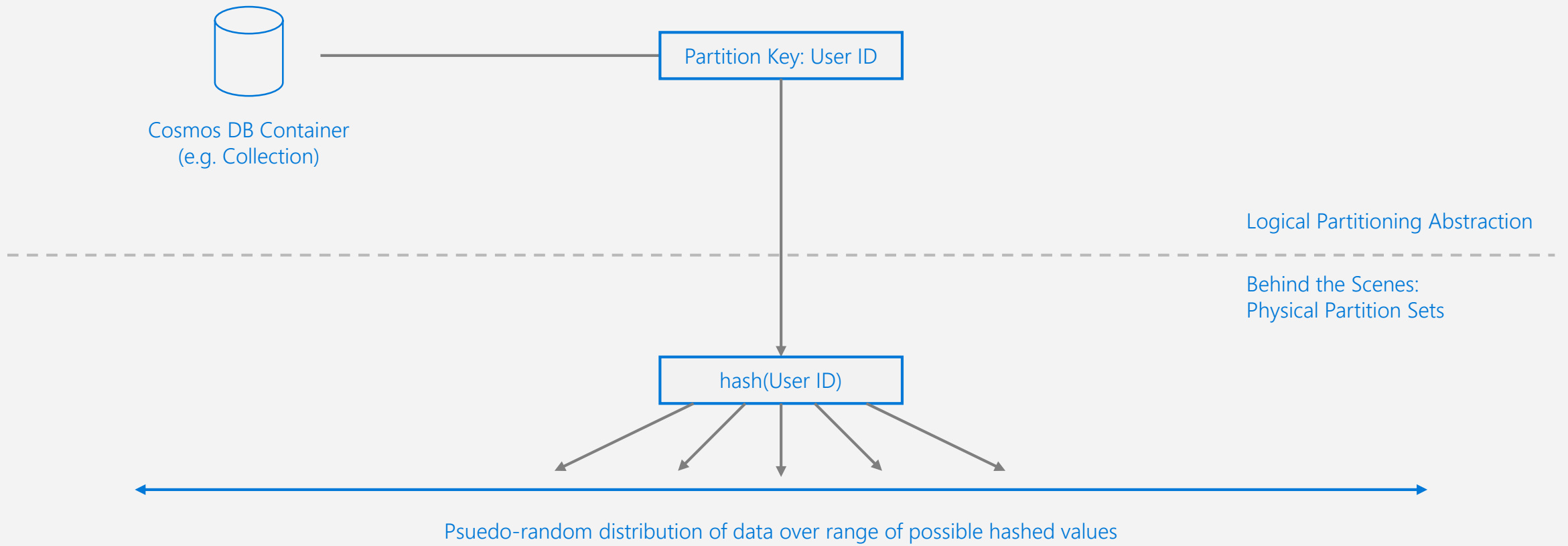
# PARTITIONING

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

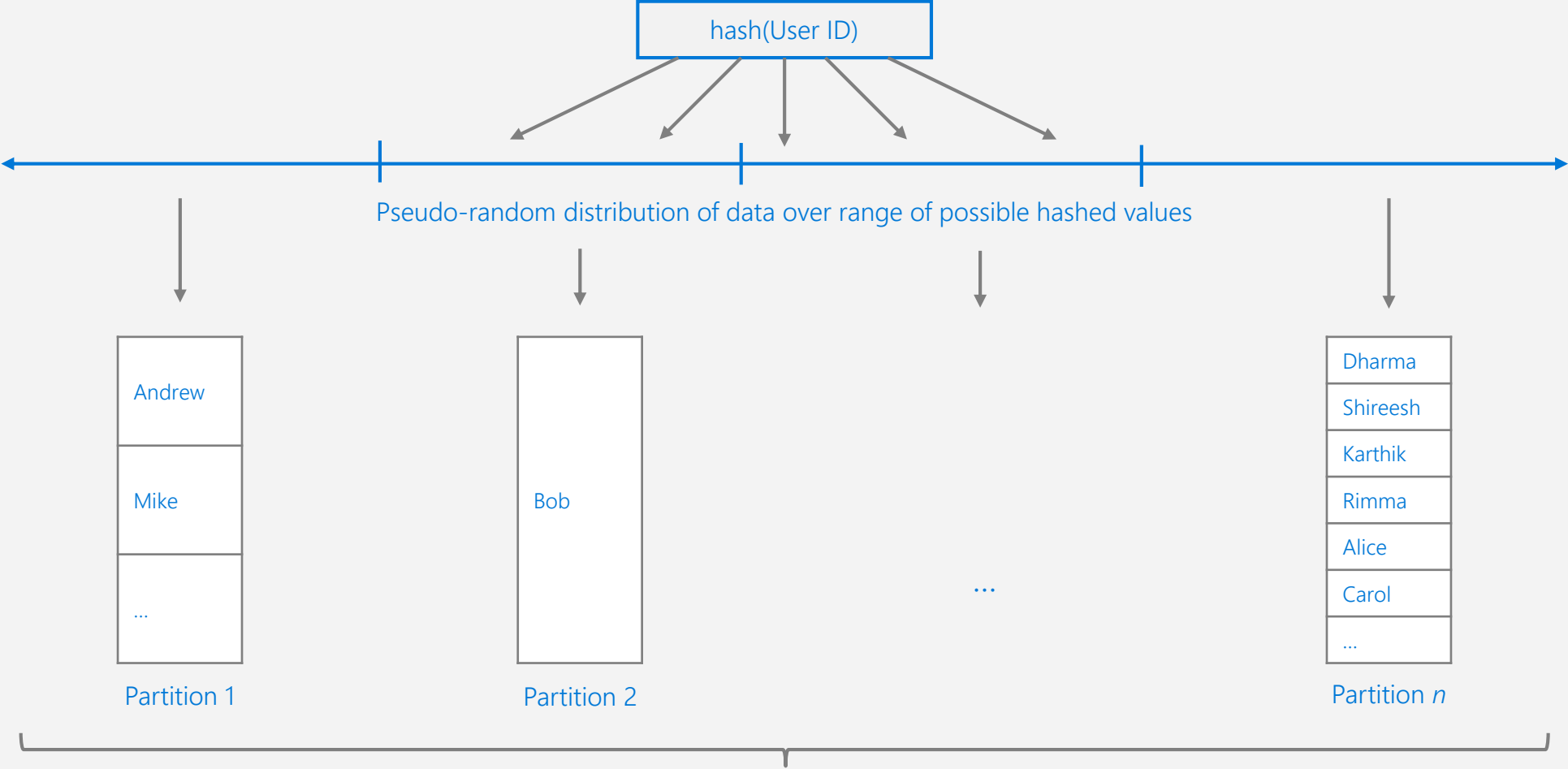
*This module will reference partitioning in the context of all Azure Cosmos DB modules and APIs.*



# PARTITIONS

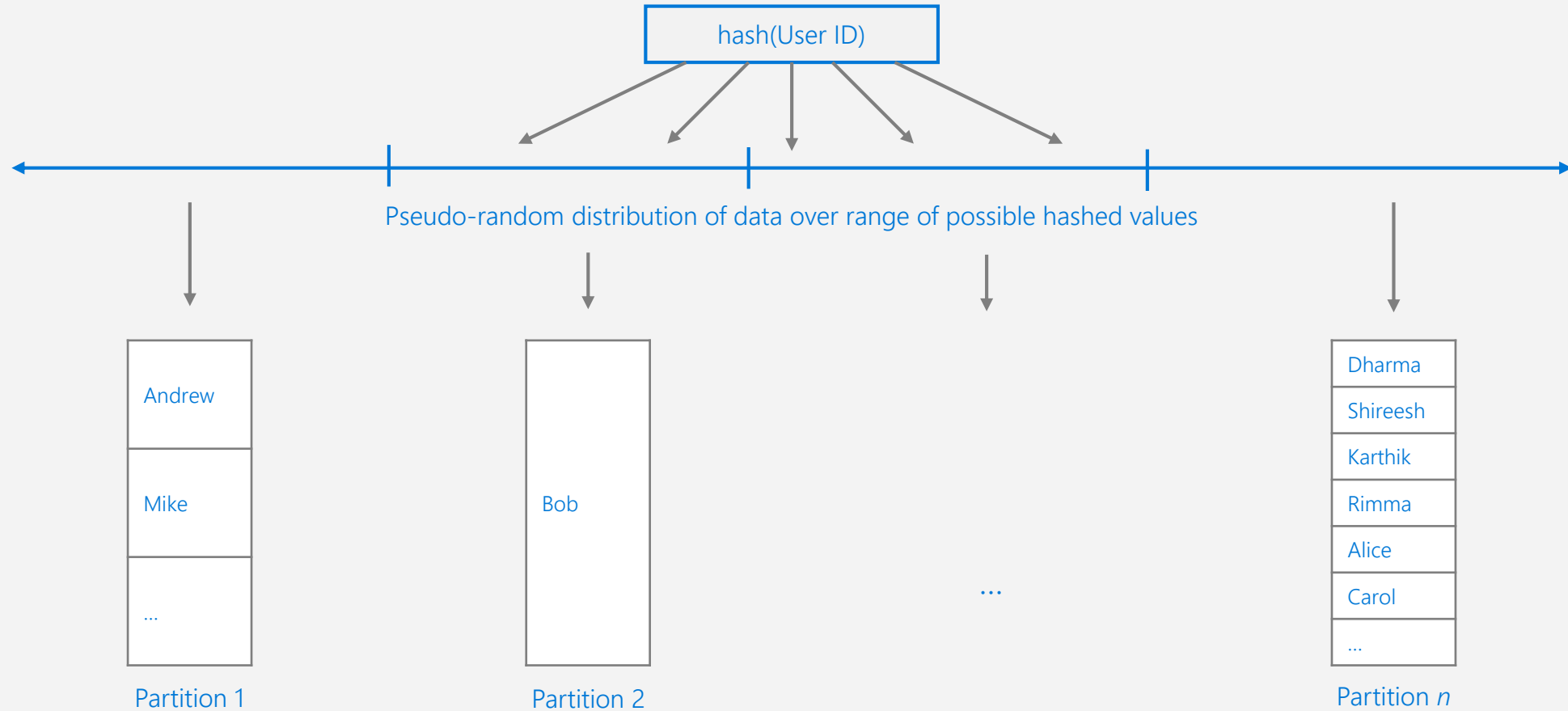


# PARTITIONS



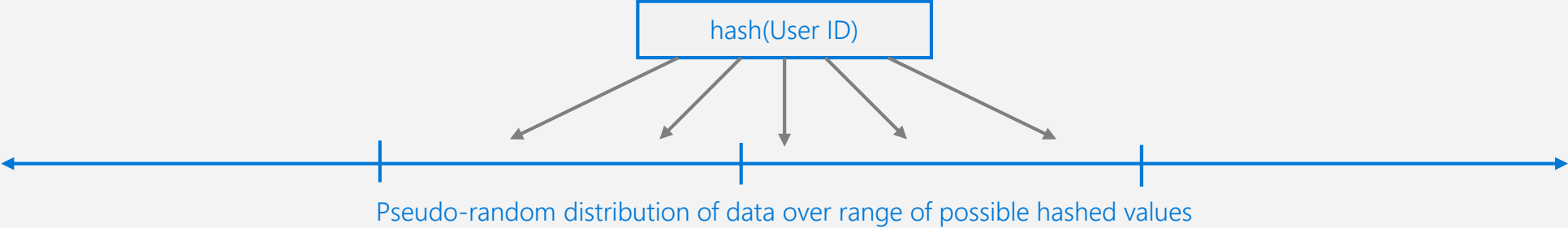
Frugal # of Partitions based on actual storage and throughput needs  
(yielding scalability with low total cost of ownership)

# PARTITIONS



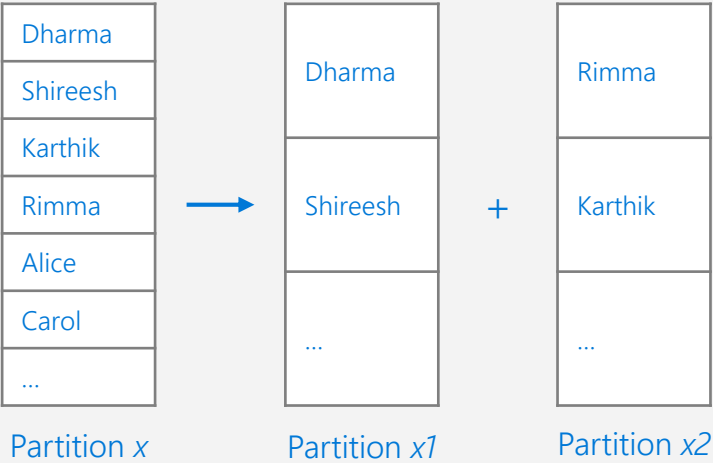
What happens when partitions need to grow?

# 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.



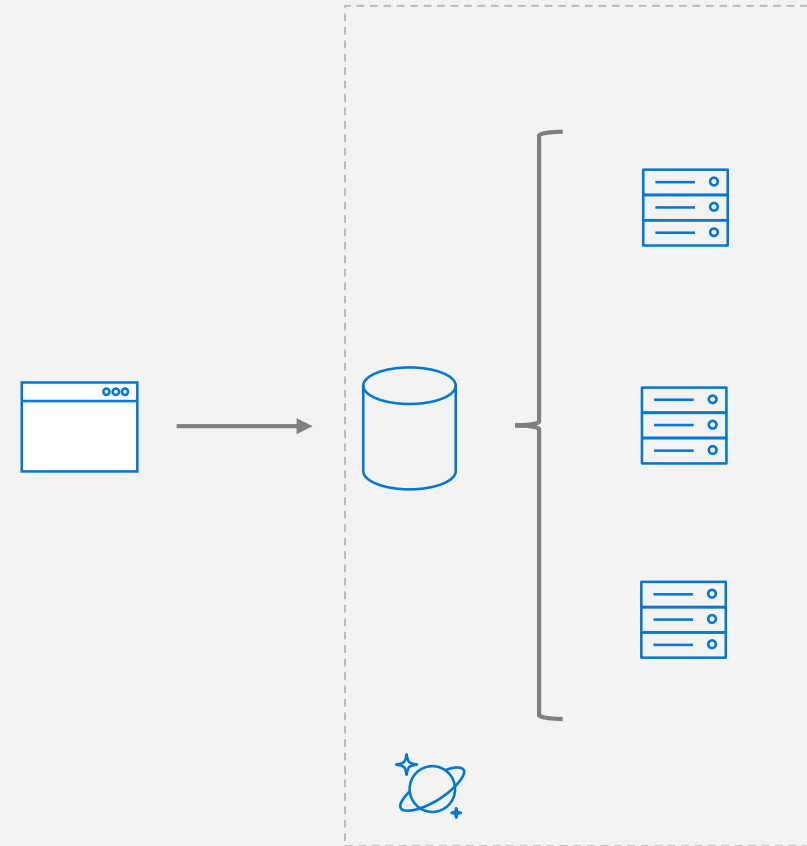
# PARTITION DESIGN

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.

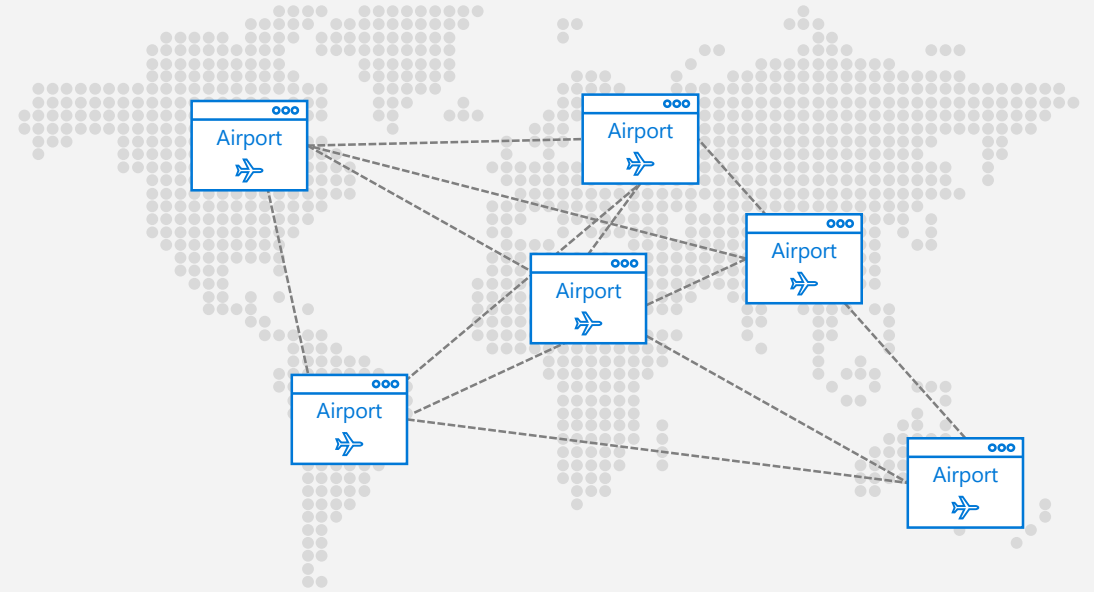
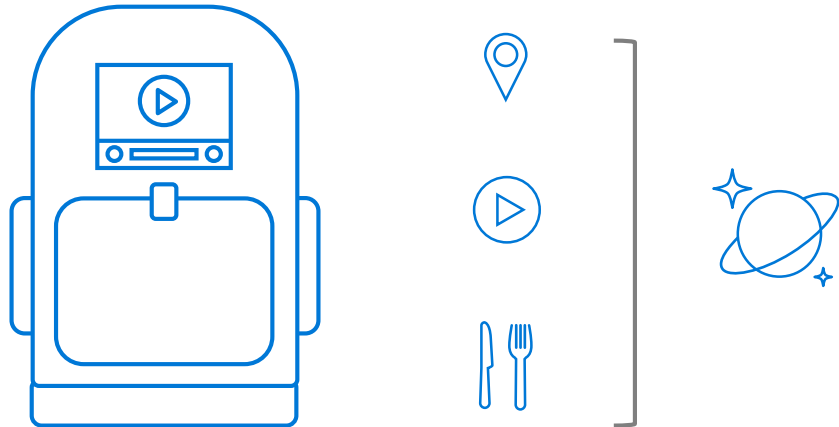
## KEY MOTIVATIONS

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



# PARTITION KEY SELECTION

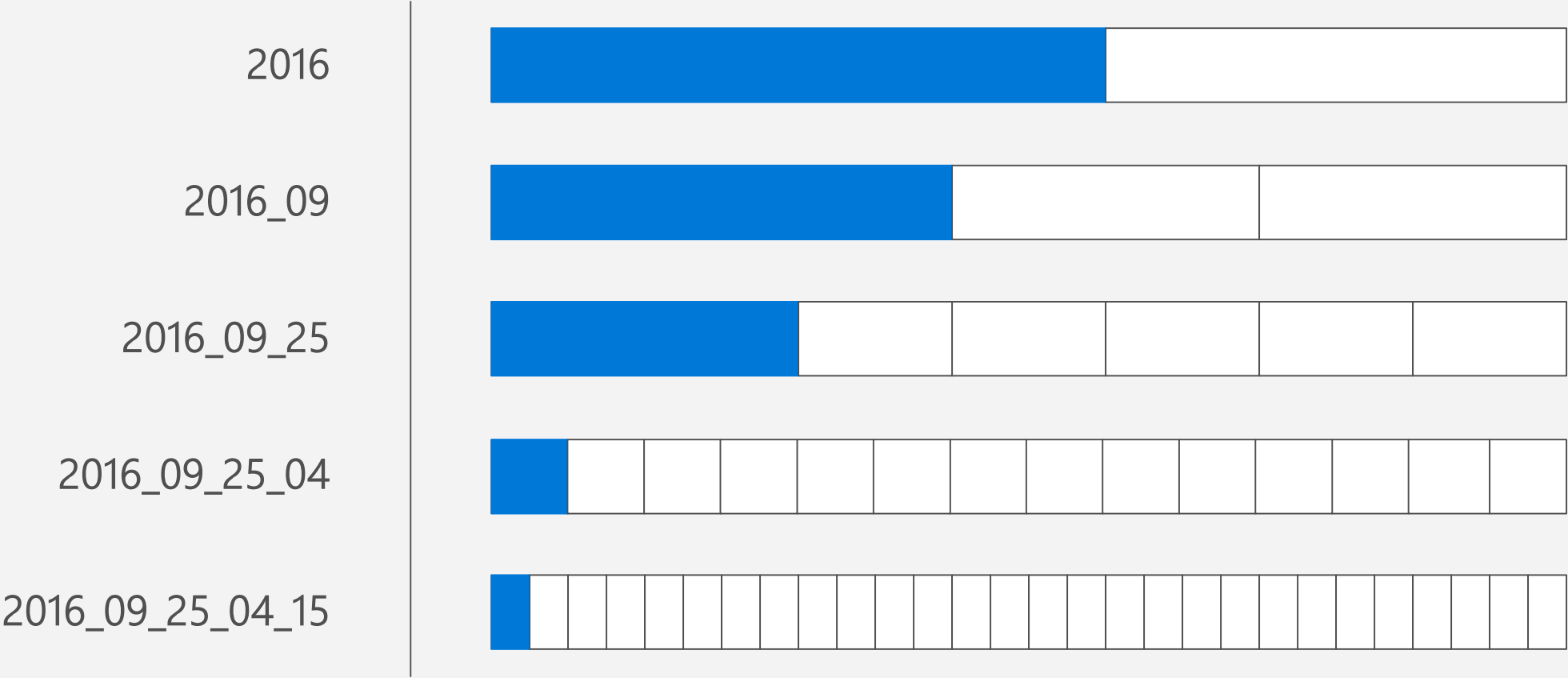
Contoso Air would like to track each customer's interactions with the in-flight entertainment system. The team has decided to partition based on the time when the interaction occurred.





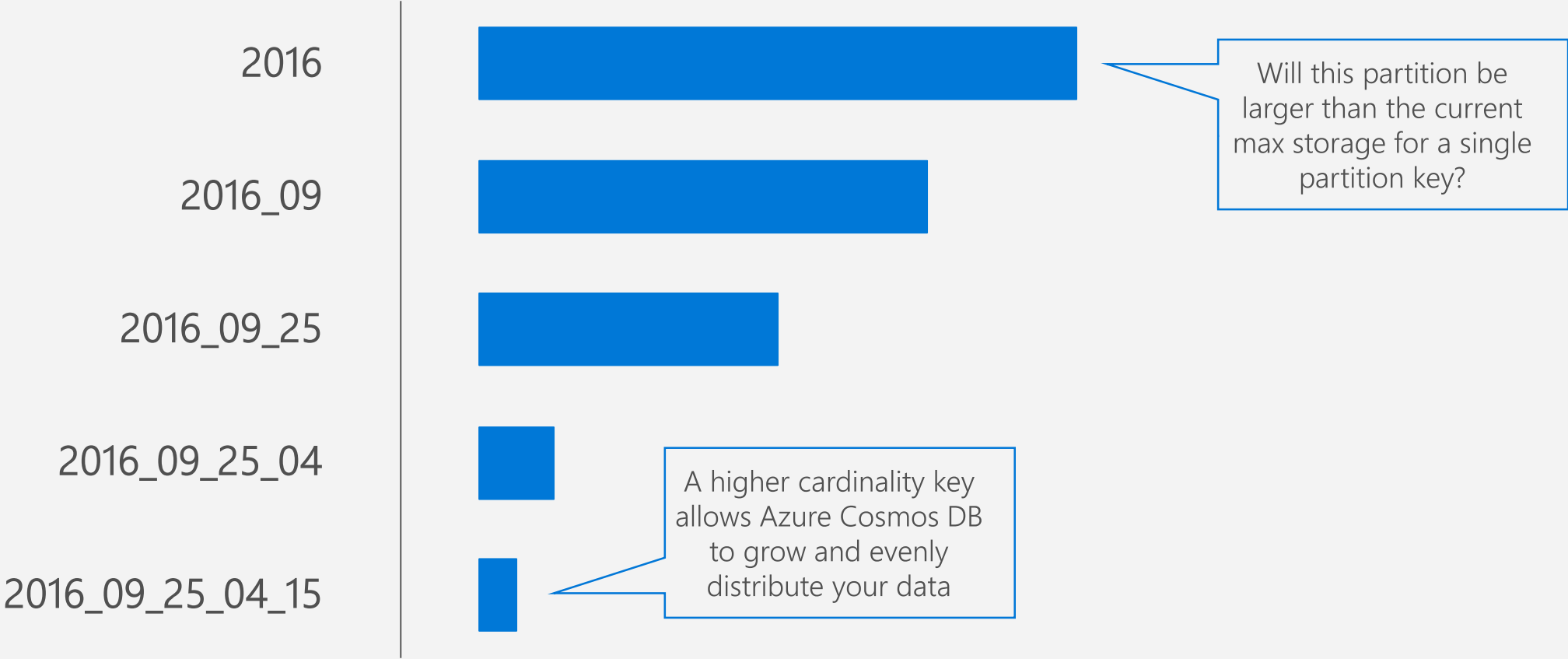
# PARTITION KEY SCENARIO

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

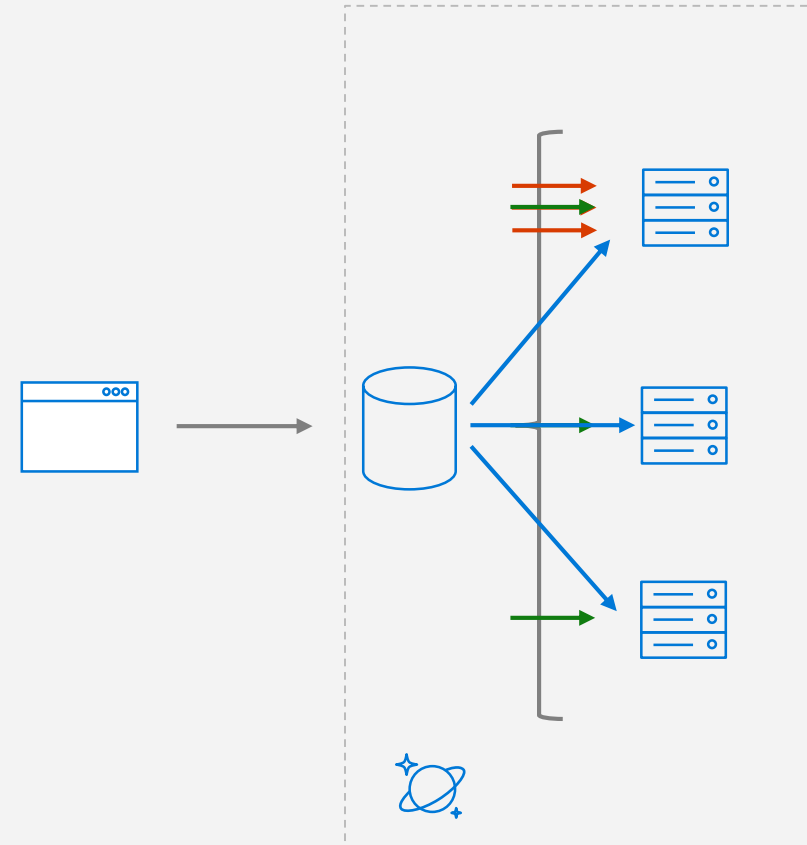
- Distribute the overall request + storage volume
  - Avoid "hot" partition keys
- Partition Key is scope for multi-record transactions and routing queries
  - Queries can be intelligently routed via partition key
  - Omitting partition key on query requires fan-out

## 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



# 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

# Managing Data

# 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



Strong



Bounded-staleness



Session



Consistent prefix



Eventual



# 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.

