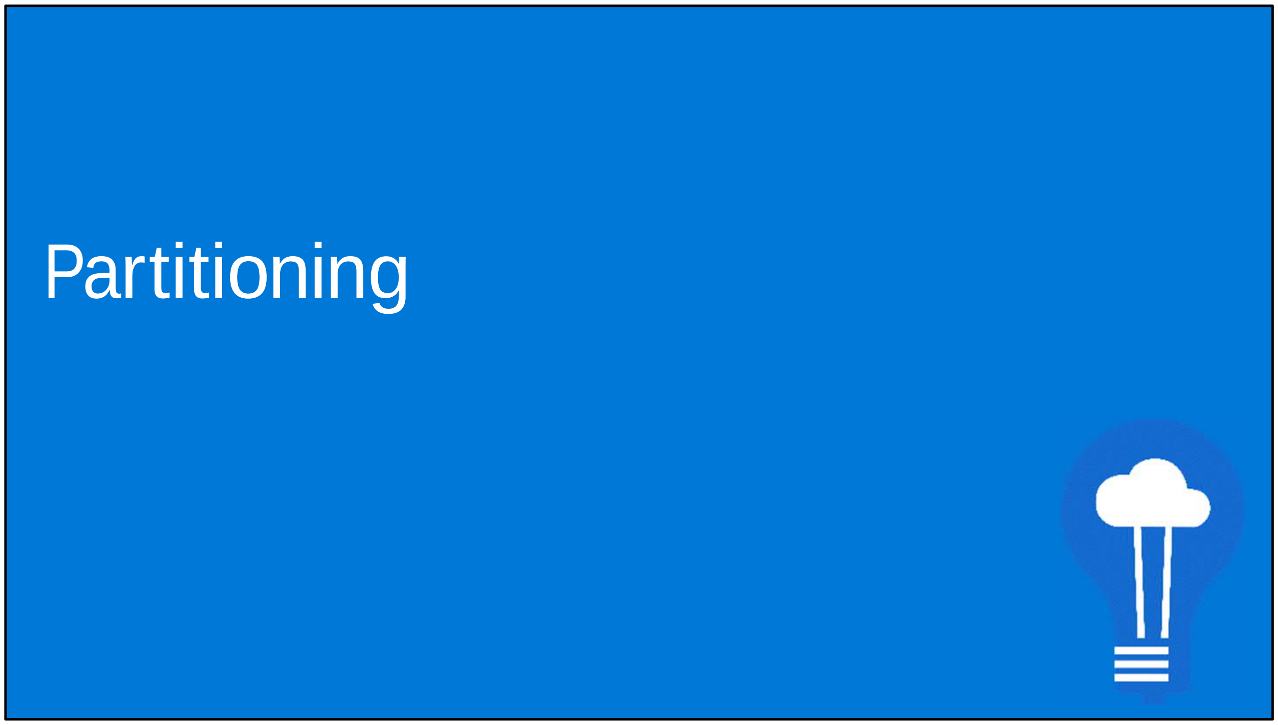
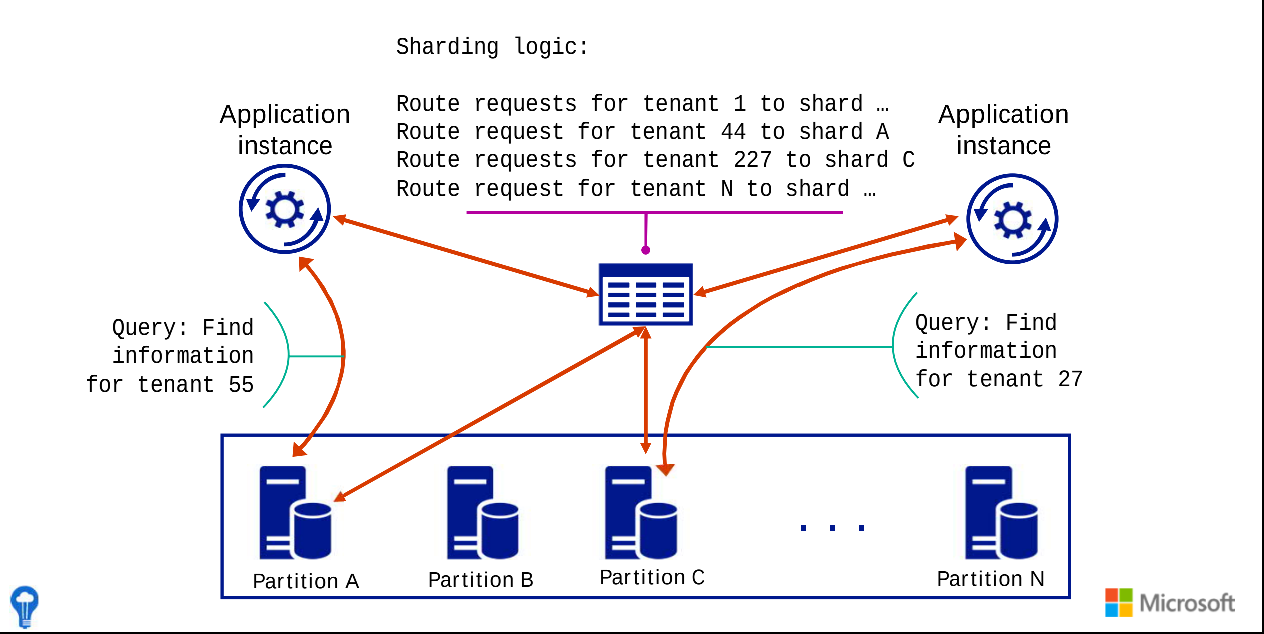
Azure Cosmos DB partition design

Azure Cosmos DB deep dive



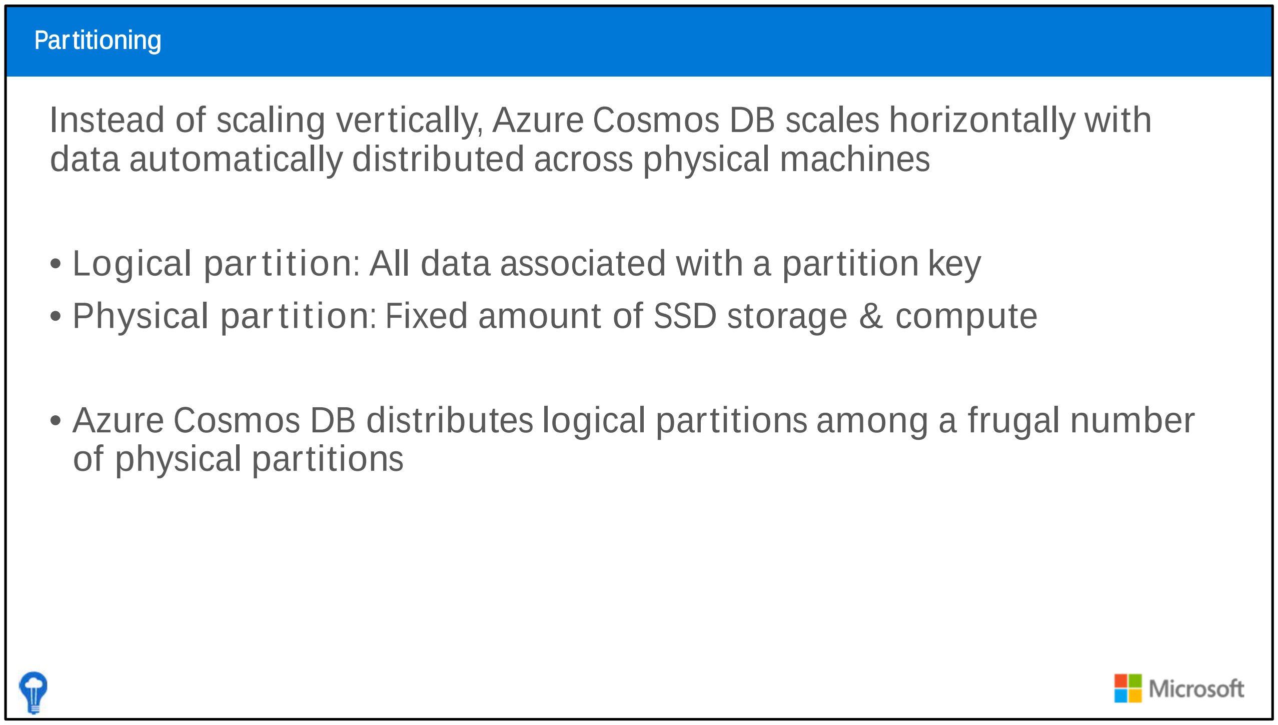
Partitioning

Reviewing partitioning (sharding)


```
Sharding logic:

Route requests for tenant 1 to shard …
Route request for tenant 44 to shard A
Route requests for tenant 227 to shard C
Route request for tenant N to shard …
```

There are multiple strategies for sharing, this is just the lookup strategy.
A map is implemented that contains lookup data mapped by shard key. With a multi-tenant application, data using the same shard key is stored in an identical shard. In this example, the tenant ID is the shard key.
https://aka.ms/Sharding_pattern



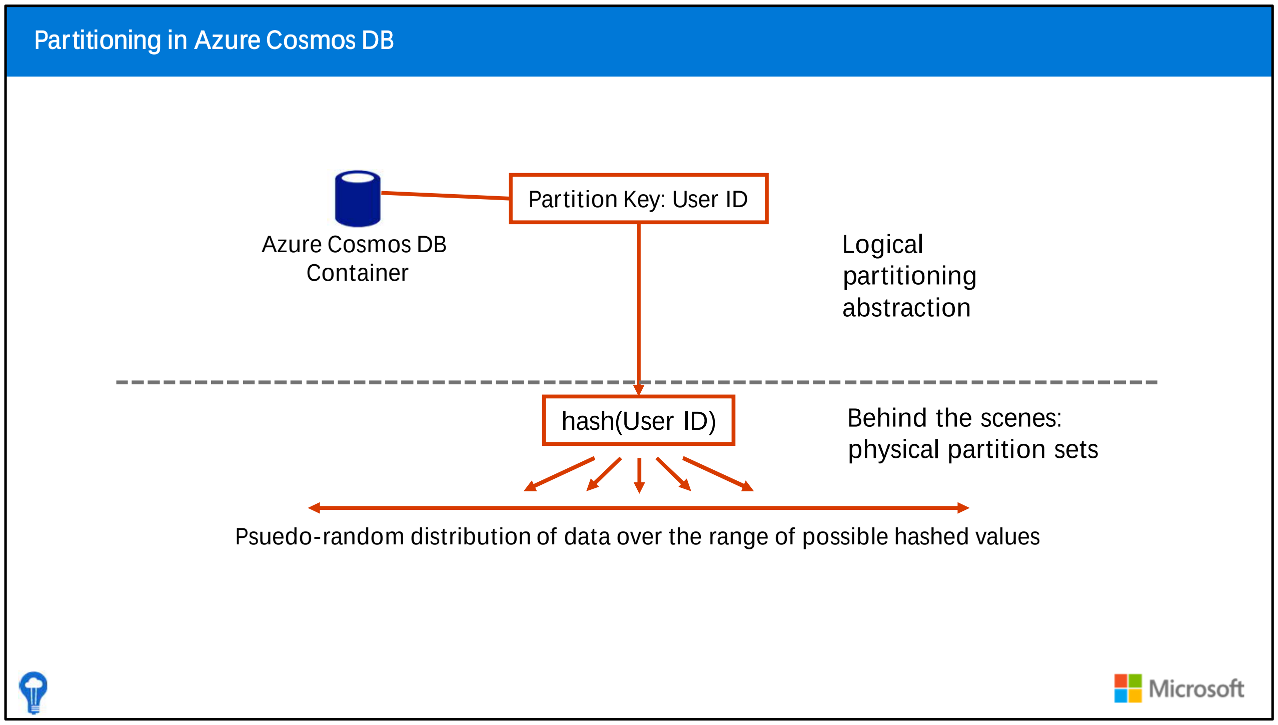
Partitioning

Instead of scaling vertically, Azure Cosmos DB scales horizontally with data automatically distributed across physical machines

- Logical partition: All data associated with a partition key
- Physical partition: Fixed amount of SSD storage & compute

- Azure Cosmos DB distributes logical partitions among a frugal number of physical partitions

From a user's perspective, logical partitions are "what you design for".
Users define one partition key per container.


Partitioning in Azure Cosmos DB
Azure Cosmos DB Container
Partition Key: User ID
Logical partitioning abstraction
hash(User ID)
Behind the scenes: physical partition sets
Psuedo-random distribution of data over the range of possible hashed values
Microsoft

Azure Cosmos DB uses partitioning to scale containers in a database to meet your application's performance needs. The items in a container are divided into distinct subsets called logical partitions. Logical partitions form based on the value of a partition key associated with each item in a container. All the items in a logical partition have the same partition key value.

For example, a container holds items. Each item has a unique value for the UserID property. If UserID serves as the partition key for the items in the container and there are 1,000 unique UserID values, 1,000 logical partitions are created for the container.

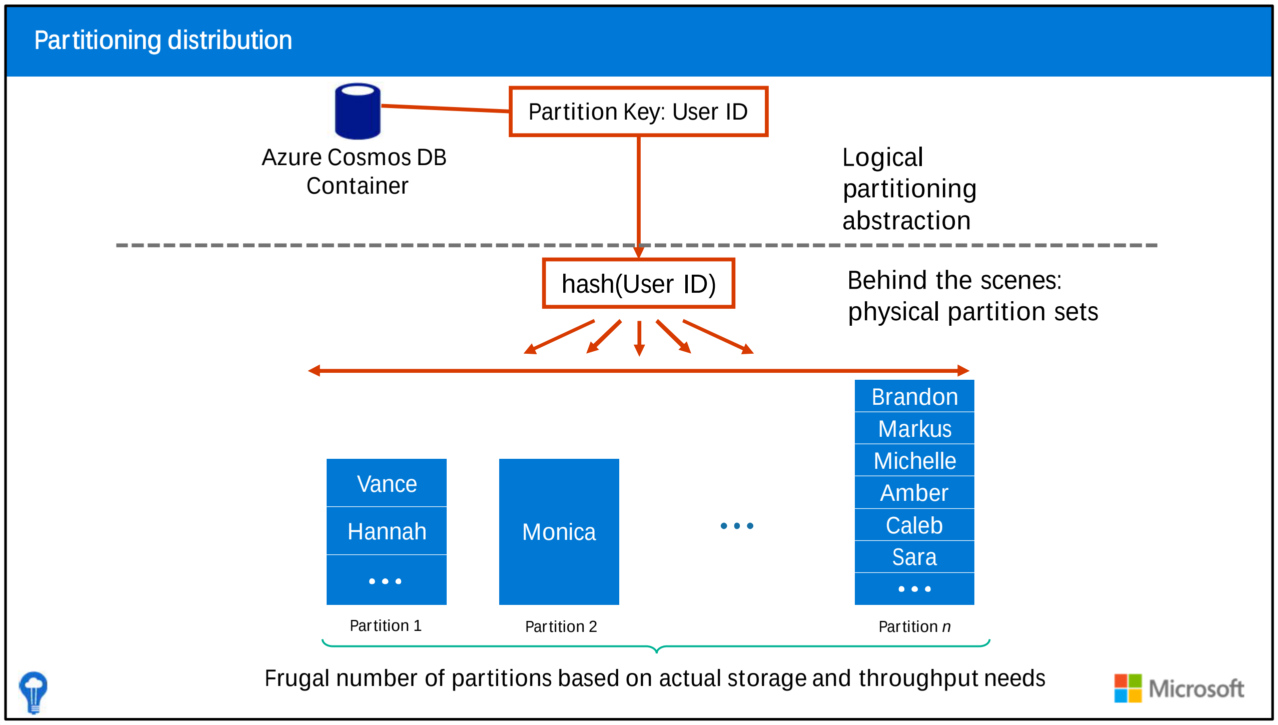
Each item in a container has a partition key that determines its logical partition and an item ID unique within that partition. Combining the partition key and the item ID creates the item's index, which uniquely identifies the item. Choosing a partition key is an important decision that affects your application's performance.

Logical partitions

A logical partition is a set of items that share the same partition key. For example, in a container that contains data about food nutrition, all items contain a foodGroup property. Use foodGroup as the partition key for the container. Groups of items that have specific values for foodGroup, such as Beef Products, Baked Products, and Sausages and Luncheon

Meats, form distinct logical partitions. When new items are added to a container, the system transparently creates new logical partitions. You don't have to worry about deleting a logical partition when the underlying data is deleted.

There's no limit to the number of logical partitions in a container. Each logical partition can store up to 20 GB of data. Effective partition keys have a wide range of possible values. For example, in a container where all items contain a foodGroup property, the data within the Beef Products logical partition can grow up to 20 GB. Selecting a partition key with a wide range of possible values ensures that the container is able to scale.


Partitioning distribution

Partition Key: User ID
Azure Cosmos DB Container
Logical partitioning abstraction
hash(User ID)
Behind the scenes: physical partition sets
Vance
Hannah
Monica
Brandon
Markus
Michelle
Amber
Caleb
Sara
Partition 1
Partition 2
Partition n
Frugal number of partitions based on actual storage and throughput needs
Microsoft

Physical partitions

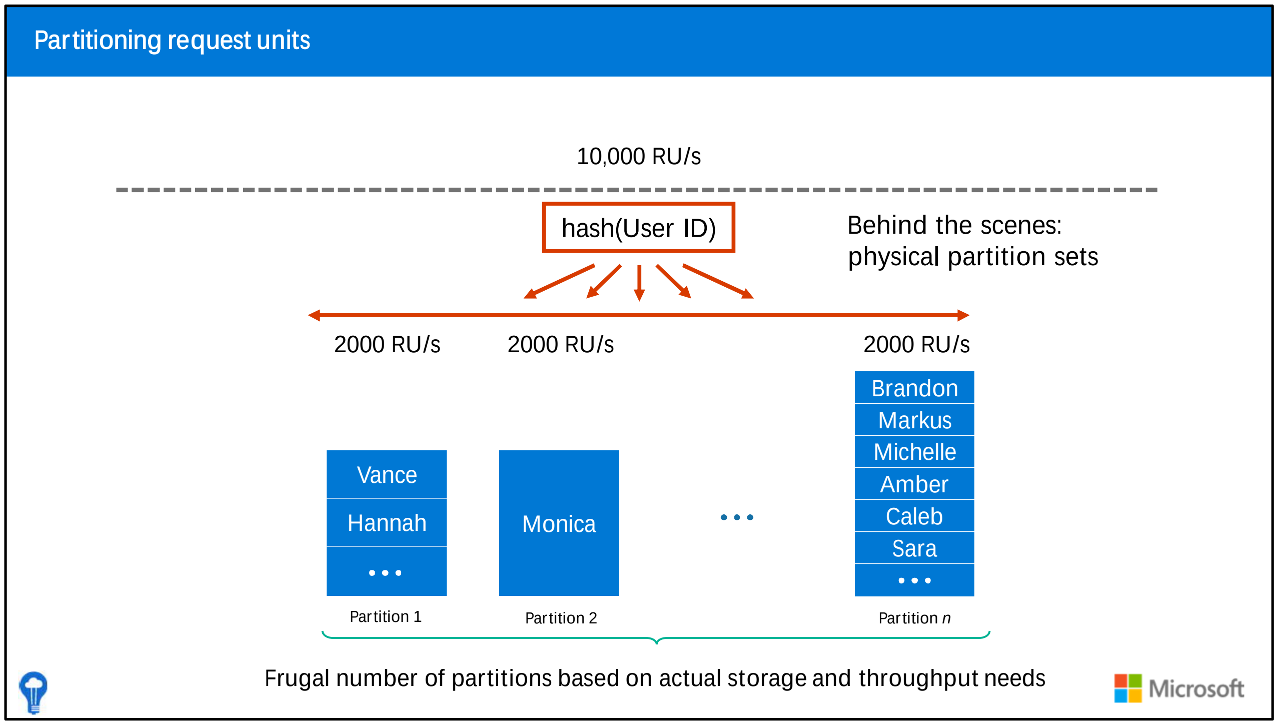
A container scales by distributing data and throughput across physical partitions. Internally, one or more logical partitions map to a single physical partition. Typically, smaller containers have many logical partitions but require only a single physical partition. Unlike logical partitions, physical partitions are an internal system implementation, and Azure Cosmos DB fully manages them.

The number of physical partitions in a container depends on these characteristics:
The amount of throughput provisioned (each individual physical partition can provide a throughput of up to 10,000 request units per second). The 10,000 RU/s limit for physical partitions implies that logical partitions also have a 10,000 RU/s limit, as each logical partition is only mapped to one physical partition.

The total data storage (each individual physical partition can store up to 50 gigabytes of data).

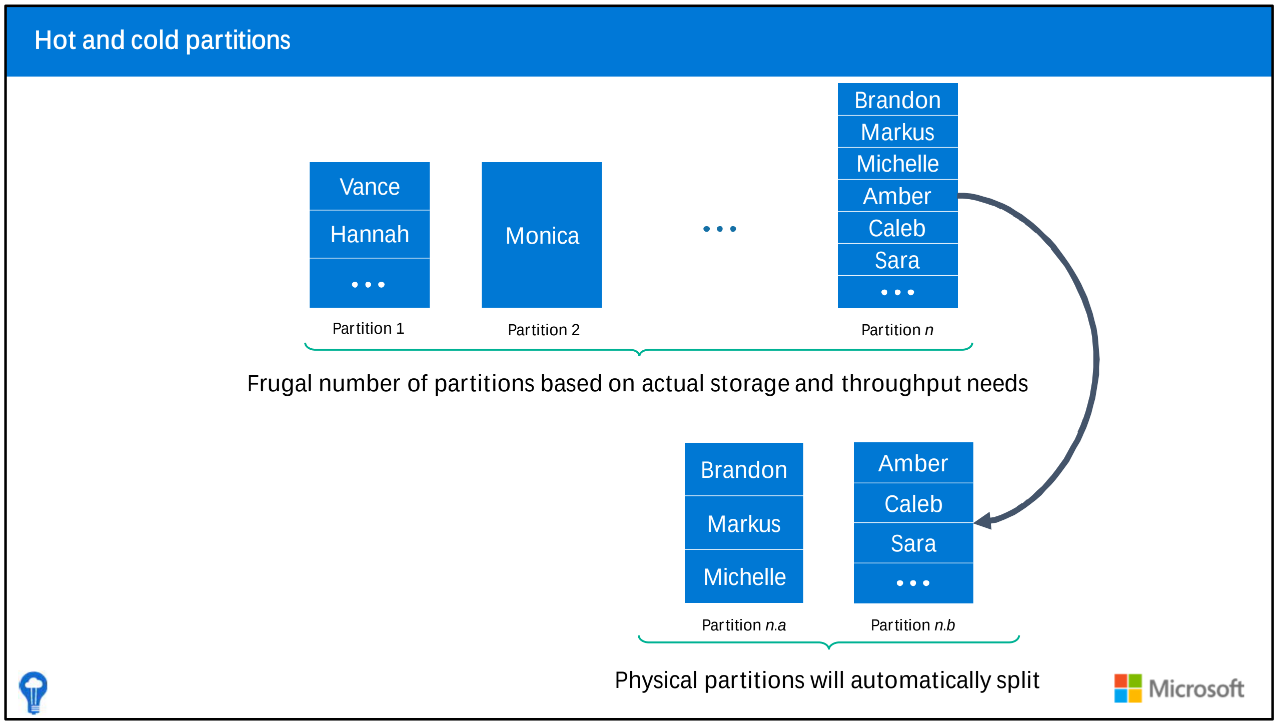
Note: Physical partitions are an internal system implementation, fully managed by Azure Cosmos DB. When developing your solutions, don't focus on physical partitions because you can't control them. Instead, focus on partition keys. Choosing a partition key that evenly distributes throughput consumption across logical partitions ensures balanced throughput consumption across physical partitions.

There's no limit to the total number of physical partitions in a container. As your provisioned throughput or data size grows, Azure Cosmos DB automatically creates new physical partitions by splitting existing ones. Physical partition splits don't affect your application's availability. After the physical partition split, all data within a single logical partition will still be stored on the same physical partition. A physical partition split simply creates a new mapping of logical partitions to physical partitions.



Partitioning request units

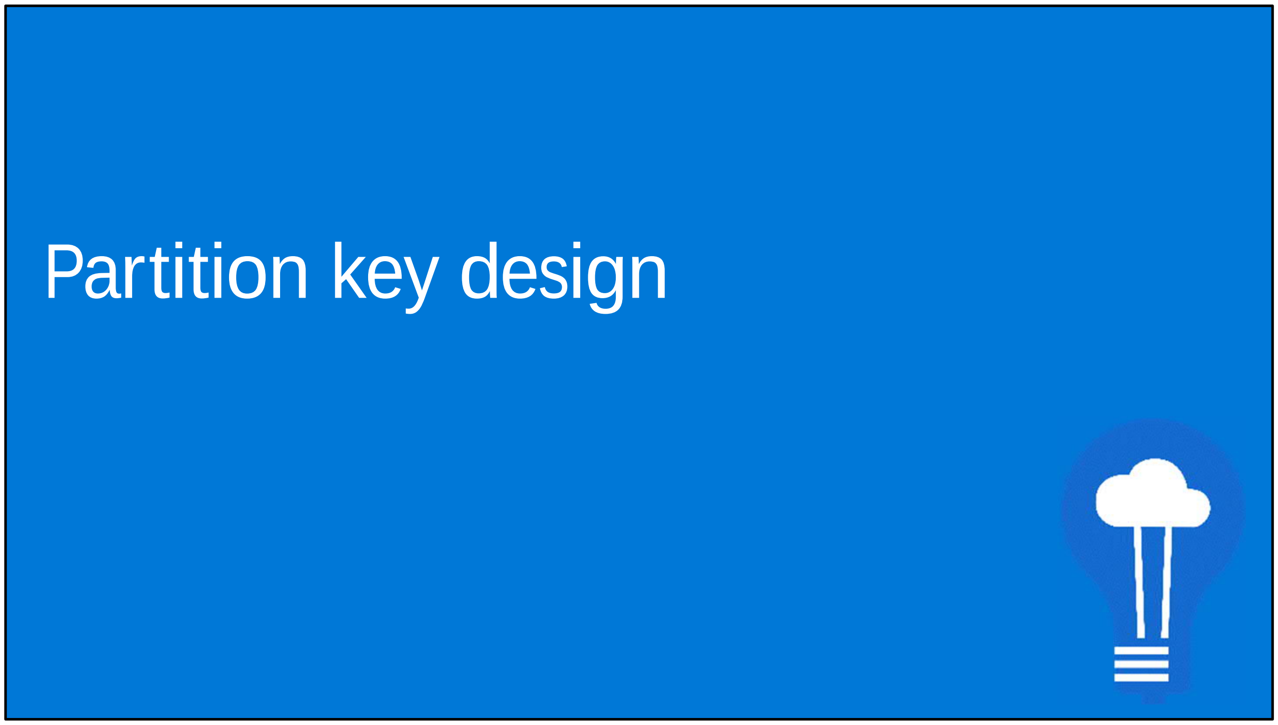
Azure Cosmos DB will automatically scale the number of physical partitions based on your workload.
A logical partition is a partition within a physical partition that stores all the data associated with a single partition key value.
https://docs.microsoft.com/en-us/azure/cosmos-db/partition-data



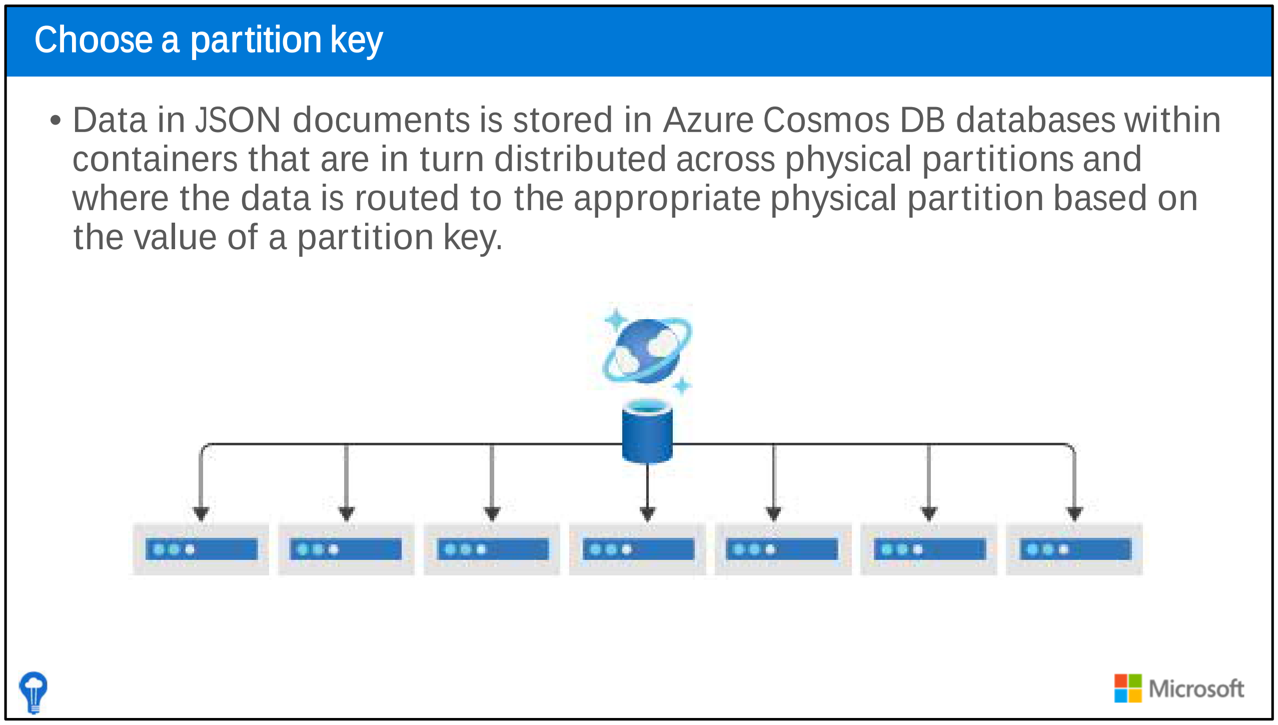
Provisioned throughput for a container divides evenly among physical partitions. A partition key design that doesn't distribute requests evenly might result in too many requests directed to a small subset of partitions that become "hot." Hot partitions cause inefficient use of provisioned throughput, which can result in rate limiting and higher costs.

For example, consider a container with the path /foodGroup specified as the partition key. The container could have any number of physical partitions, but in this example we assume it has three. A single physical partition could contain multiple partition keys. As an example, the largest physical partition could contain the top three most significant size logical partitions: Beef Products, Vegetable and Vegetable Products, and Soups, Sauces, and Gravies.

If you assign a throughput of 18,000 request units per second (RU/s), each of the three physical partitions uses one-third of the total provisioned throughput. Within the selected physical partition, the logical partition keys Beef Products, Vegetable and Vegetable Products, and Soups, Sauces, and Gravies can, collectively, utilize the physical partition's 6,000 provisioned RU/s. Because provisioned throughput is evenly divided across your container's physical partitions, it's important to choose a partition key that evenly distributes throughput consumption. For more information, see Choosing the right logical partition key.



Partition key design



Choose a partition key

- Data in JSON documents is stored in Azure Cosmos DB databases within containers that are in turn distributed across physical partitions and where the data is routed to the appropriate physical partition based on the value of a partition key.

https://learn.microsoft.com/en-us/training/modules/implement-non-relational-data-model/6-choose-partition-key

Choose a partition key
A partition key has two components: partition key path and the partition key value. For example, consider an item { "userId" : "Andrew", "worksFor": "Microsoft" } if you choose "userId" as the partition key, the following are the two partition key components:

The partition key path (For example: "/userId"). The partition key path accepts alphanumeric and underscores (_) characters. You can also use nested objects by using the standard path notation(/).

The partition key value (For example: "Andrew"). The partition key value can be of string or numeric types.

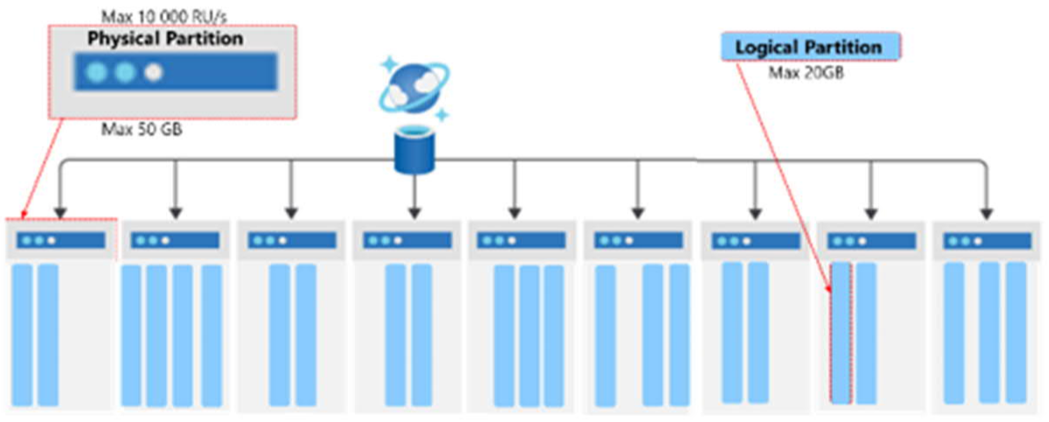
Selecting your partition key is a simple but important design choice in Azure Cosmos DB. Once you select your partition key, you can't change it in place. If you need to change your partition key, move your data to a new container with your desired partition key. Container copy jobs help with this process.

Remember that data in JSON documents is stored in Azure Cosmos DB databases within containers that are in turn distributed across physical partitions and where the data is routed to the appropriate physical partition based on the value of a partition key.

The partition key is a required document property that ensures documents with the same partition key value are routed to and stored within a specific physical partition. A physical partition supports a fixed maximum amount of storage and throughput (RU/s). Azure Cosmos DB automatically distributes the logical partitions across the available physical partitions, again using the partition key value to do so in a predictable way.

Logical partitions in Azure Cosmos DB

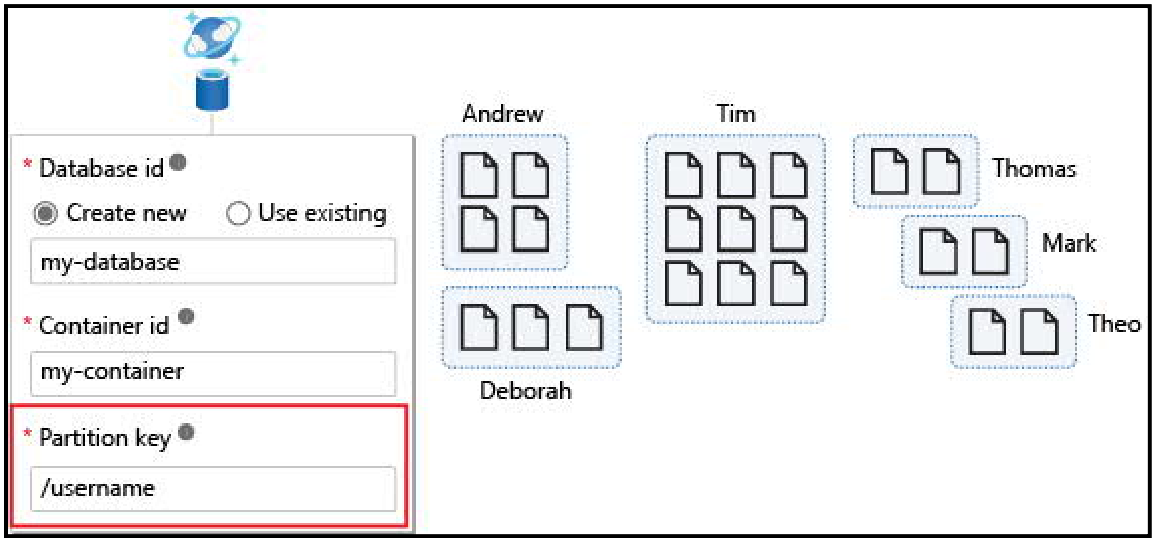
- A partition key provides a way to route data for a logical partition.
- It's a property that exists within every document in your container that routes your data.



Microsoft

A container is another abstraction for all data stored with the same partition key. The partition key is defined when you create a container.

Partition Key (Example)



* Database id

Create new [✓] Use existing

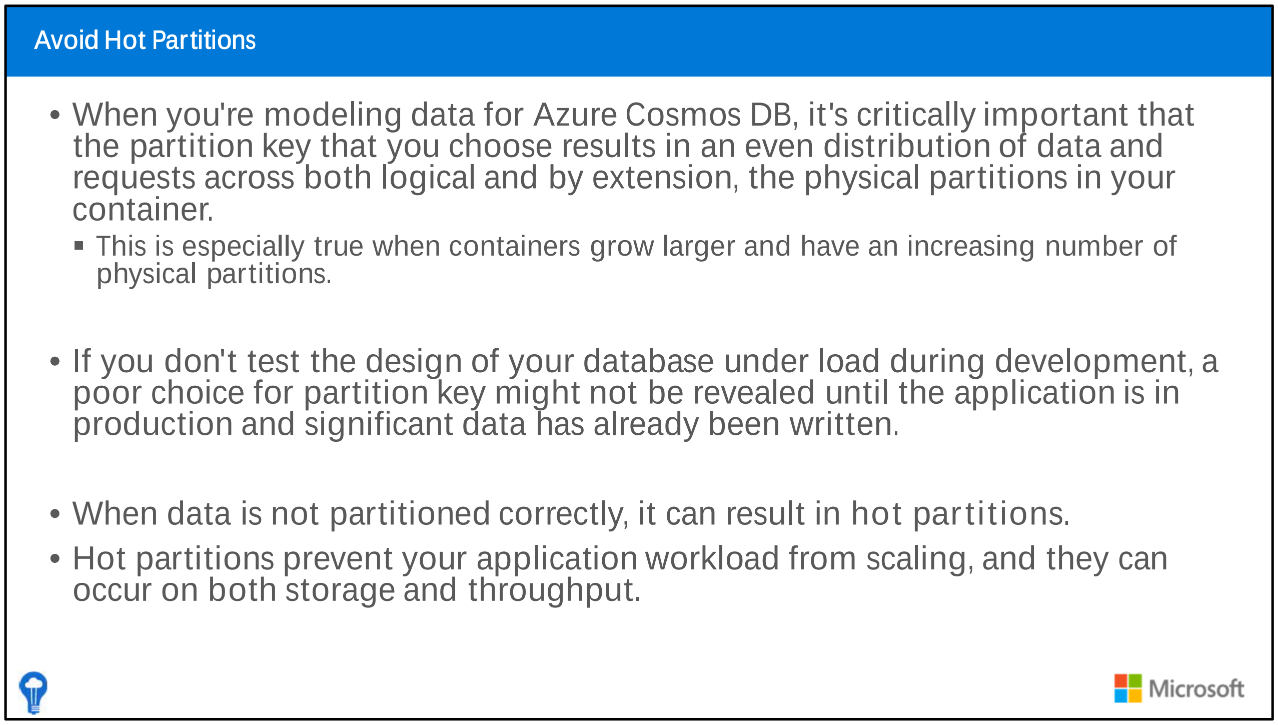
my-database

* Container id

my-container

* Partition key

/username

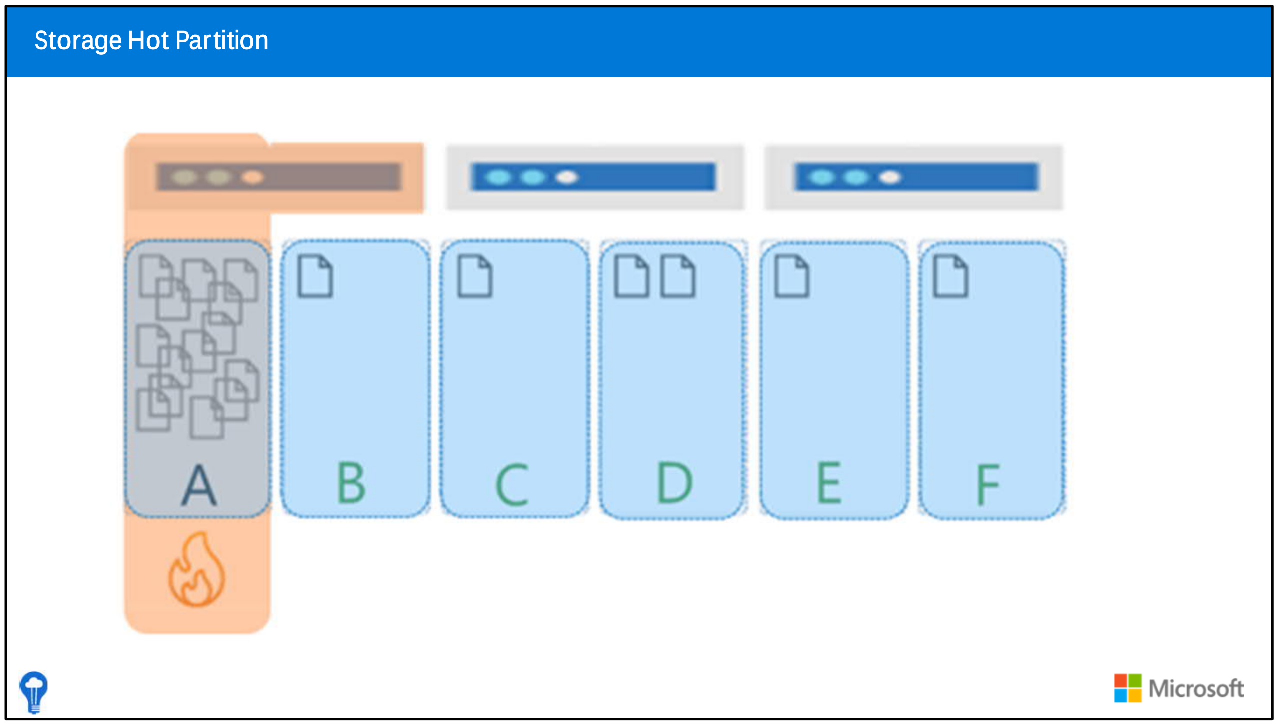


Avoid Hot Partitions

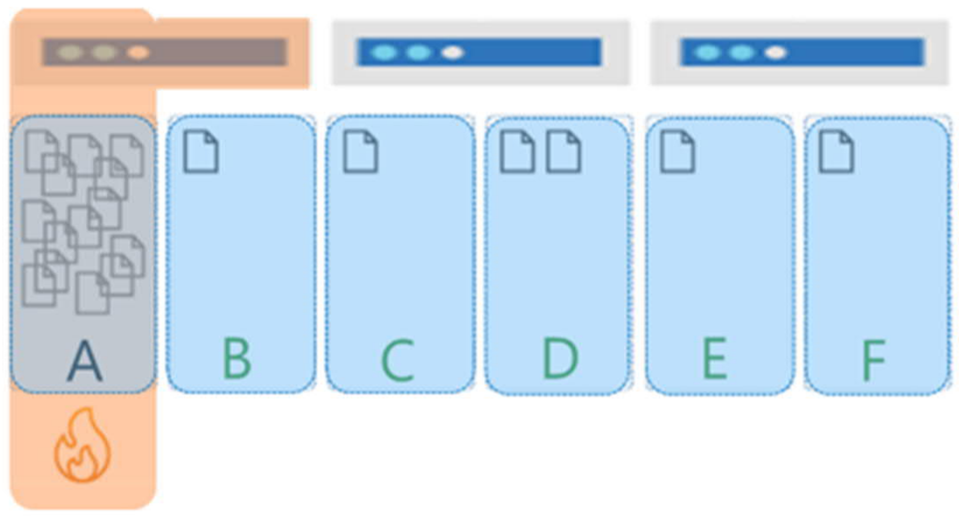
- When you're modeling data for Azure Cosmos DB, it's critically important that the partition key that you choose results in an even distribution of data and requests across both logical and by extension, the physical partitions in your container.
  - This is especially true when containers grow larger and have an increasing number of physical partitions.
- If you don't test the design of your database under load during development, a poor choice for partition key might not be revealed until the application is in production and significant data has already been written.
- When data is not partitioned correctly, it can result in hot partitions.
- Hot partitions prevent your application workload from scaling, and they can occur on both storage and throughput.

Microsoft

https://learn.microsoft.com/en-us/training/modules/implement-non-relational-data-model/6-choose-partition-key

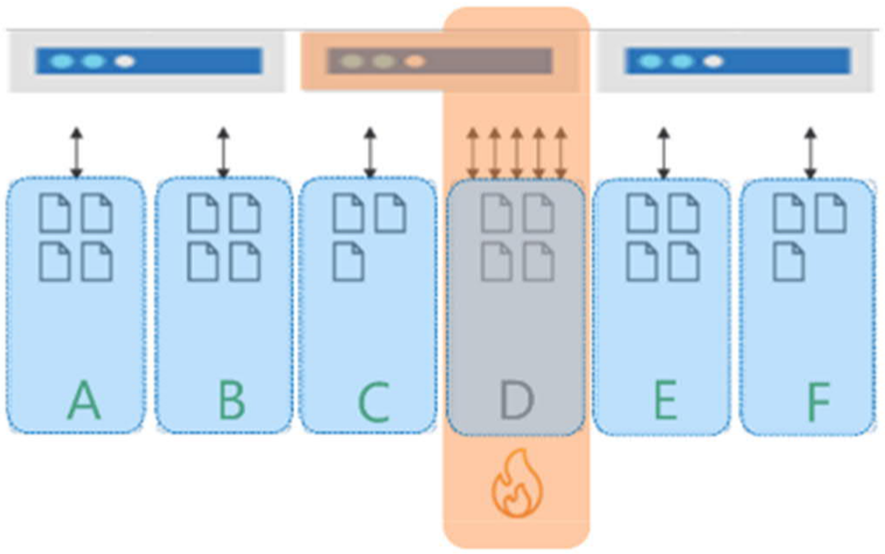


Storage Hot Partition



A hot partition on storage occurs when you have a partition key that results in highly asymmetric storage patterns. As an example, consider a multitenant application that uses TenantId as its partition key with six tenants: A to F. Tenants B,C,E and F are very small, Tenant D has a little more data. However Tenant A is massive and quickly hits the 20-GB limit for its partition. In this scenario, we need to select a different partition key that will spread the storage across more logical partitions.

Throughput hot partitions

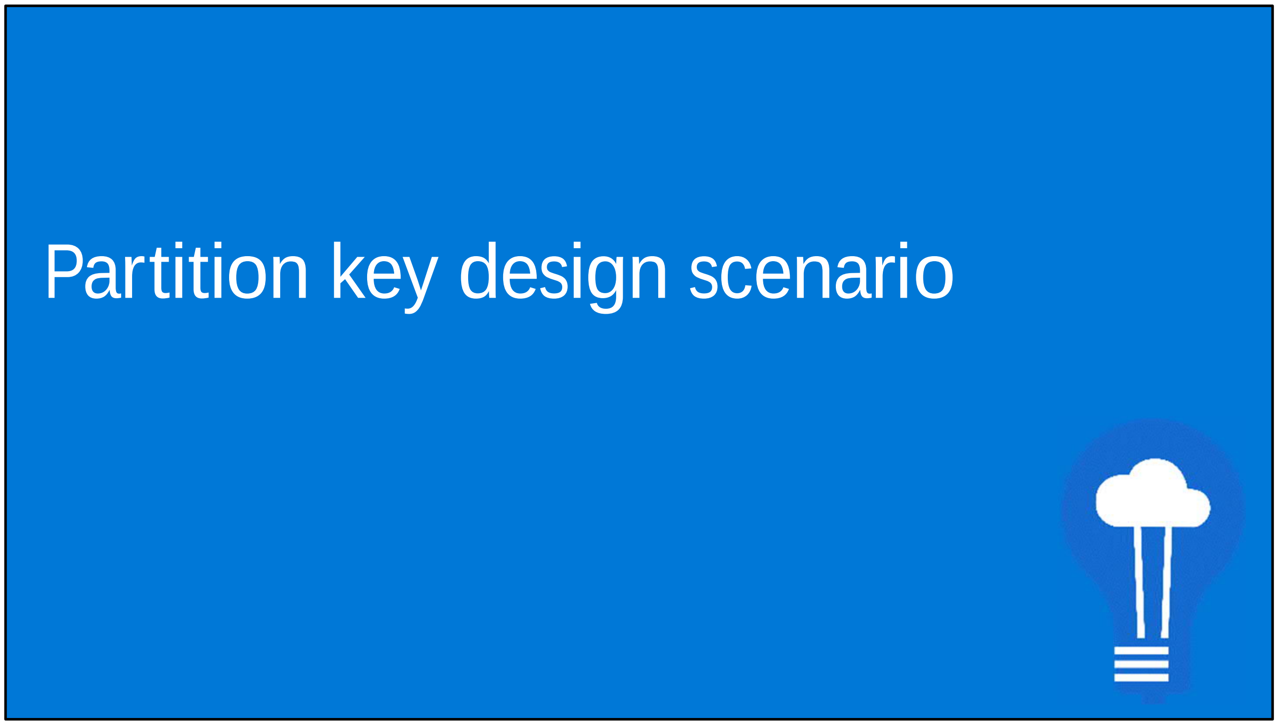


Microsoft

Throughput can suffer from hot partitions when most or all of the requests go to the same logical partition.
It's important to understand the access patterns for your application to ensure that requests are spread as evenly as possible across partition key values. When throughput is provisioned for a container in Azure Cosmos DB, it's allocated evenly across all the physical partitions within a container.

As an example, if you have a container with 30,000 RU/s, this workload is spread across the three physical partitions for the same six tenants mentioned earlier. So each physical partition gets 10,000 RU/s. If tenant D consumes all of its 10,000 RU/s, it will be rate limited because it can't consume the throughput allocated to the other partitions. This results in poor performance for tenant C and D, and leaving unused compute capacity in the other physical partitions and remaining tenants. Ultimately, this partition key results in a database design where the application workload can't scale.

When data and requests are spread evenly, the database can grow in a way that fully utilizes both the storage and throughput. The result will be the best possible performance and highest efficiency. In short, the database design will scale.

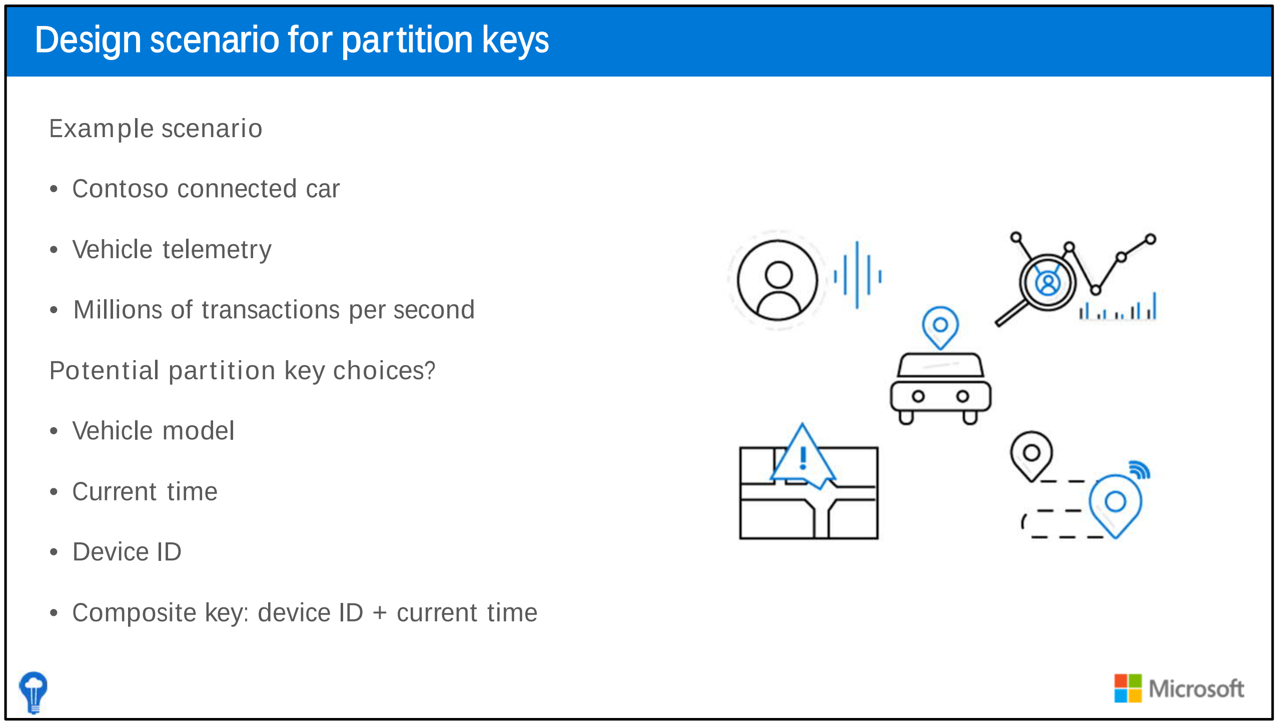


Partition key design scenario

Key design tips for partitions

- Estimate scale needs using known size and throughput
- Study the existing workload to identify common queries and operations
- Measure quantity of reads per second vs. writes per second
- Validate using a proof of concept

Microsoft



Design scenario for partition keys

Example scenario

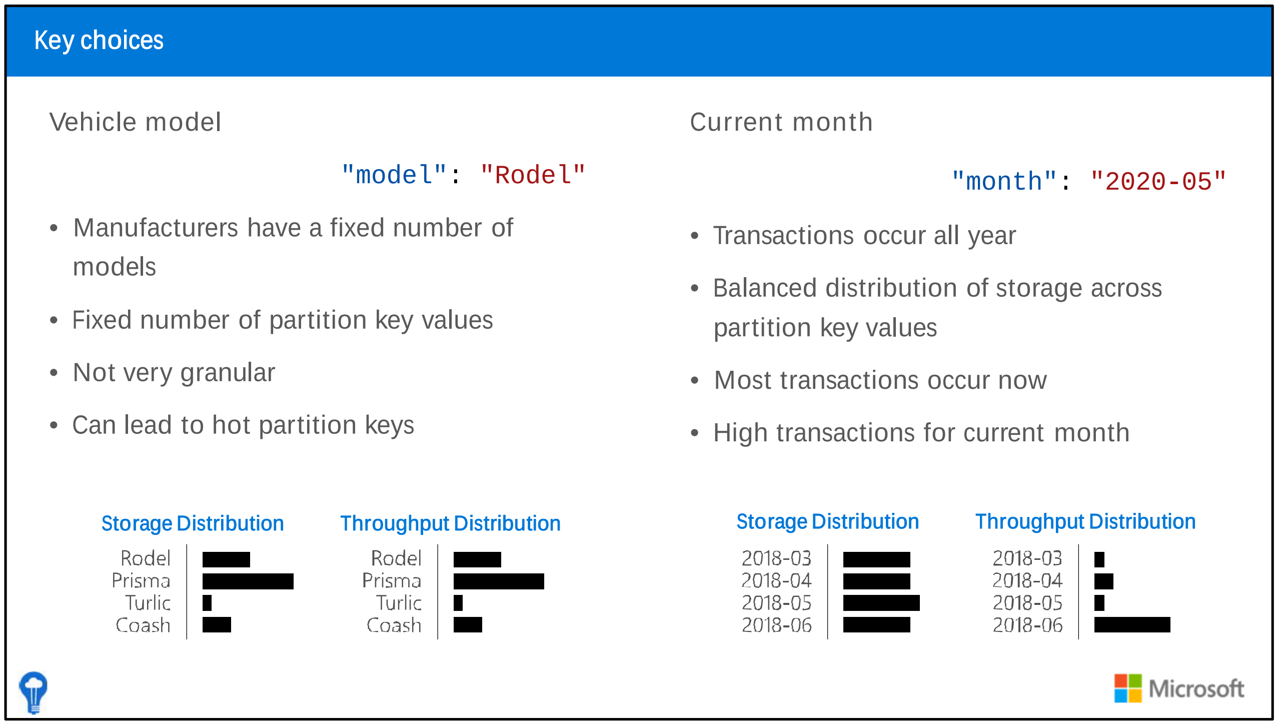
- Contoso connected car
- Vehicle telemetry
- Millions of transactions per second

Potential partition key choices?

- Vehicle model
- Current time
- Device ID
- Composite key: device ID + current time

Microsoft

Contoso Connected Car is a vehicle telematics company.
They’re 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.



Key choices

Vehicle model

```
"model": "Rodel"
```

- Manufacturers have a fixed number of models
- Fixed number of partition key values
- Not very granular
- Can lead to hot partition keys

Storage Distribution

Rodel
Prisma
Turlic
Coash

Throughput Distribution

Rodel
Prisma
Turlic
Coash

Current month

```
"month": "2020-05"
```

- Transactions occur all year
- Balanced distribution of storage across partition key values
- Most transactions occur now
- High transactions for current month

Storage Distribution

2018-03
2018-04
2018-05
2018-06

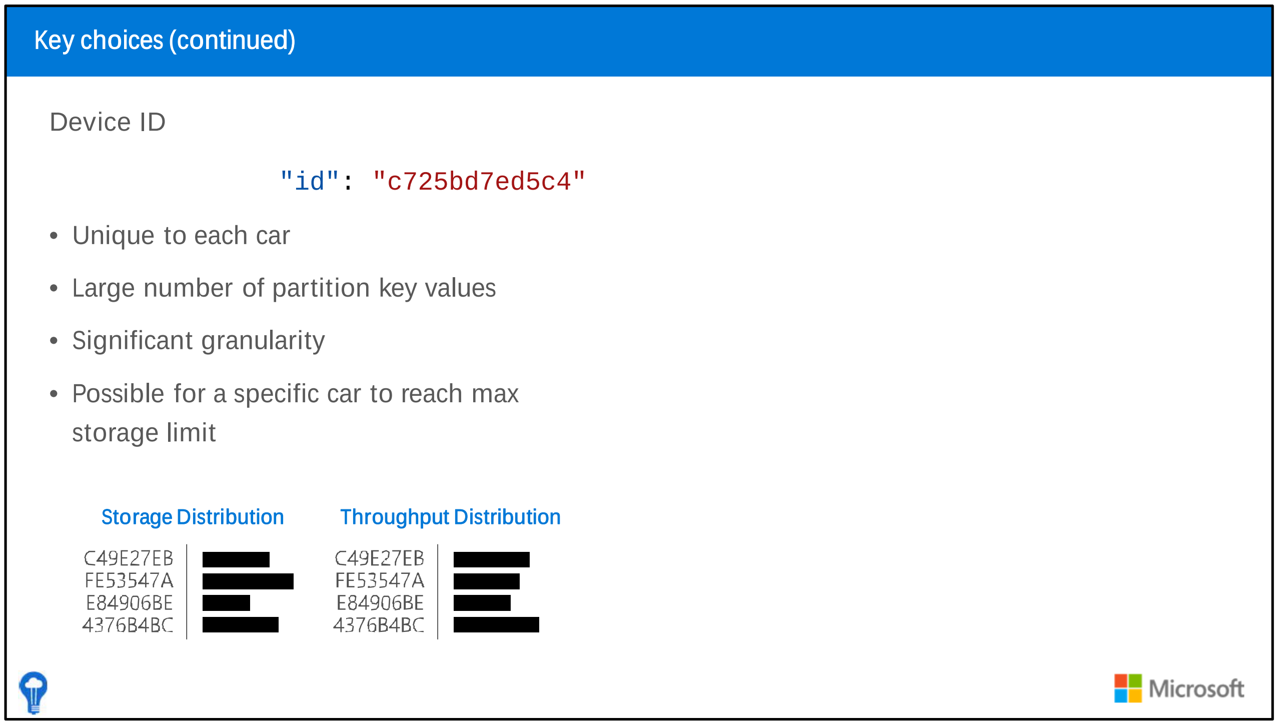
Throughput Distribution

2018-03
2018-04
2018-05
2018-06

Microsoft

Most auto manufactures only have a couple dozen models.
This creates 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.

Auto manufacturers have transactions that occur throughout the year.
This creates a more balanced distribution of storage across partition key values.
However, most business transactions occur on recent data, which creates the possibility of a hot partition key for the current month on throughput.

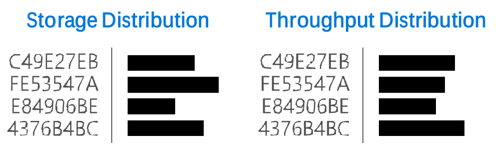


Key choices (continued)

Device ID

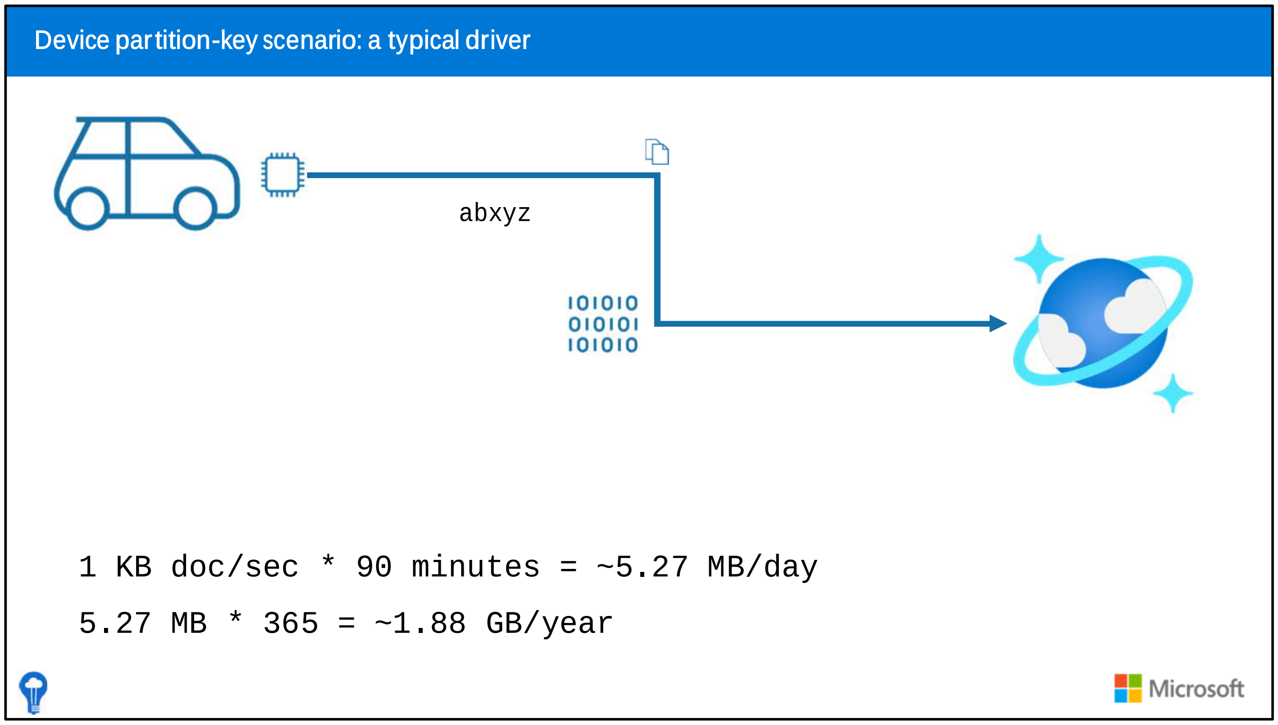
```
"id": "c725bd7ed5c4"
```

- Unique to each car
- Large number of partition key values
- Significant granularity
- Possible for a specific car to reach max storage limit

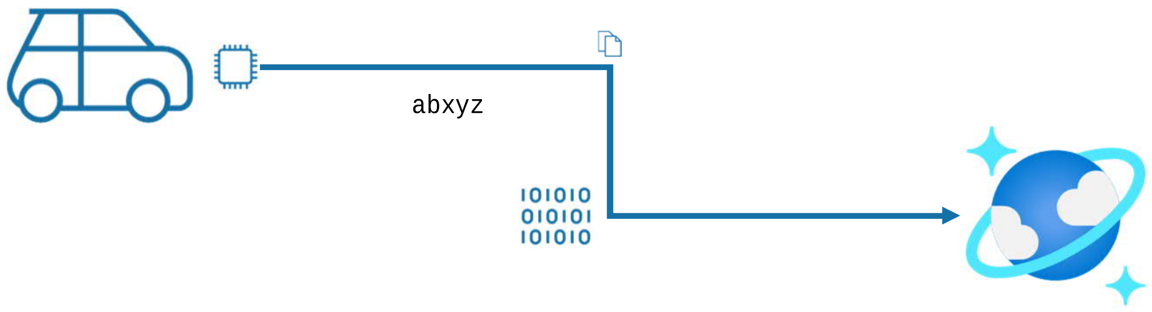


Microsoft

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's possible that a specific partition key will reach the storage limit per partition key.

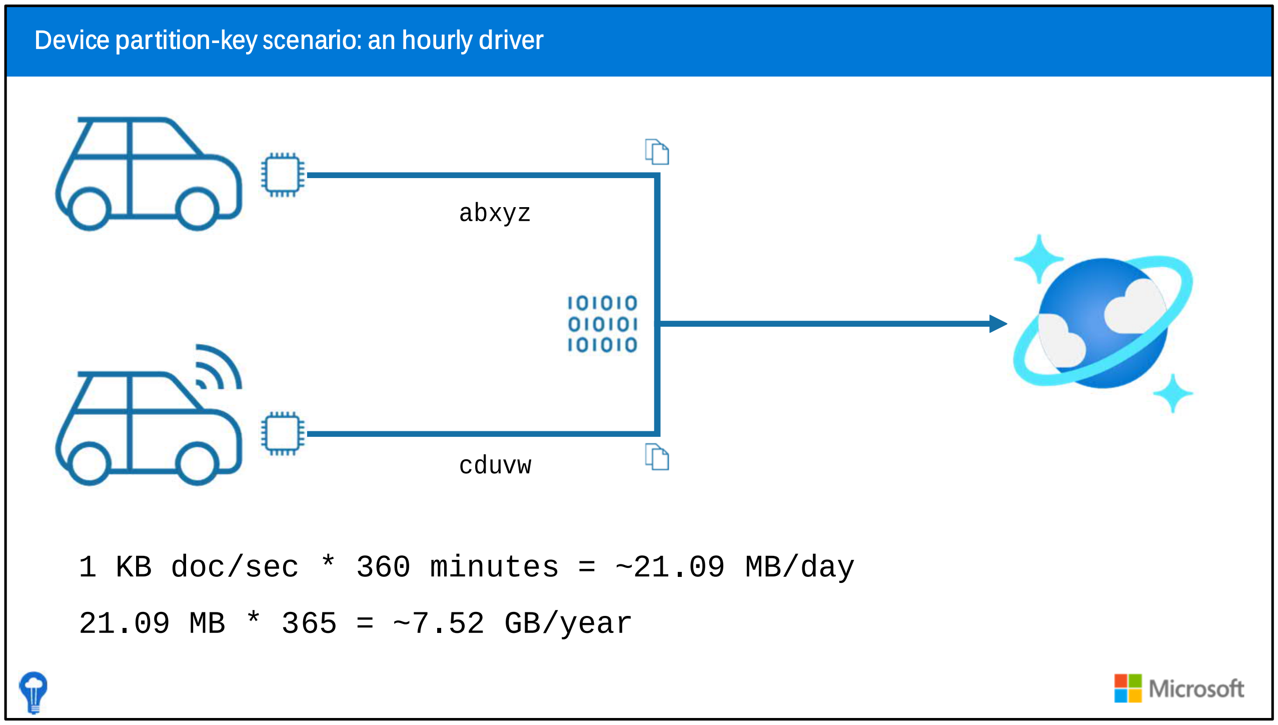


Device partition-key scenario: a typical driver

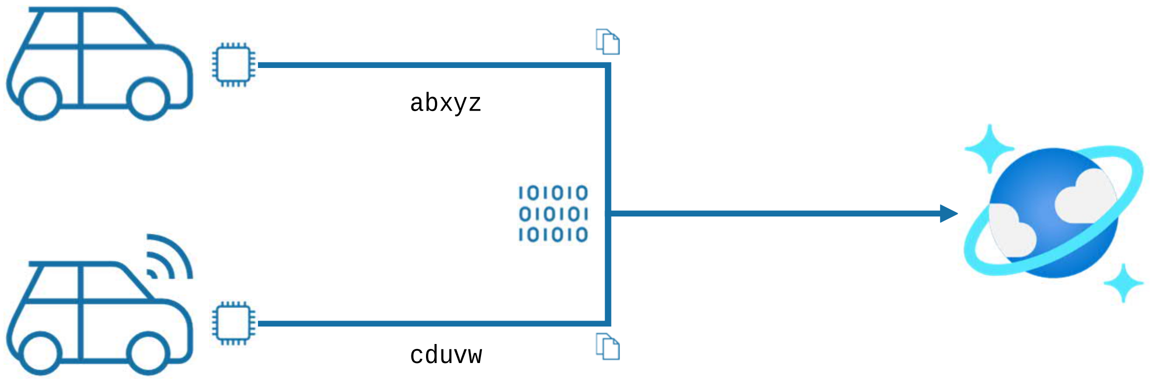


```
1 KB doc/sec * 90 minutes = ~5.27 MB/day
5.27 MB * 365 = ~1.88 GB/year
```

For this example, let's assume a 1 kilobyte (KB) JSON telemetry document is created per second because that’s the baseline for the service-level agreement (SLA) guarantees.
A typical driver drives 90 minutes a day and would generate approximately 2 gigabytes (GBs) of telemetry data per year.
While the driver might exceed the 20 GB max per logical partition, it’s exceedingly rare. Many developers will choose this as an appropriate level of granularity.



Device partition-key scenario: an hourly driver

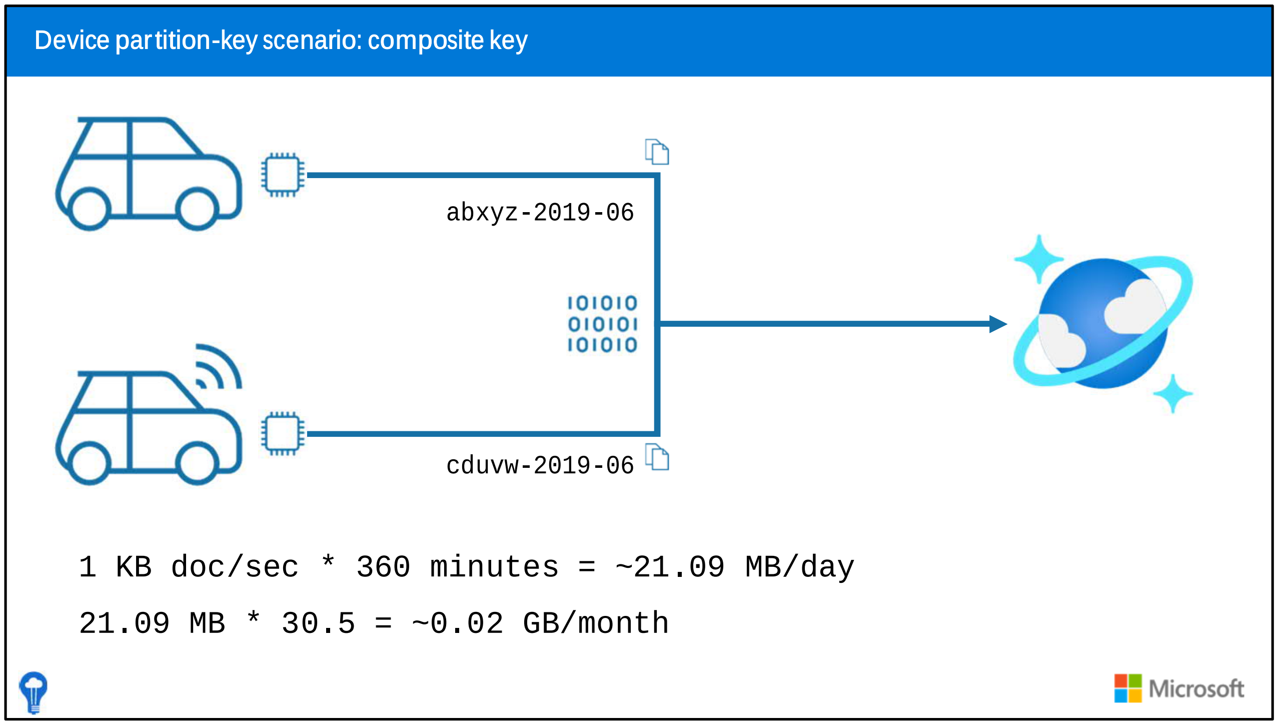


```
1 KB doc/sec * 360 minutes = ~21.09 MB/day
21.09 MB * 365 = ~7.52 GB/year
```

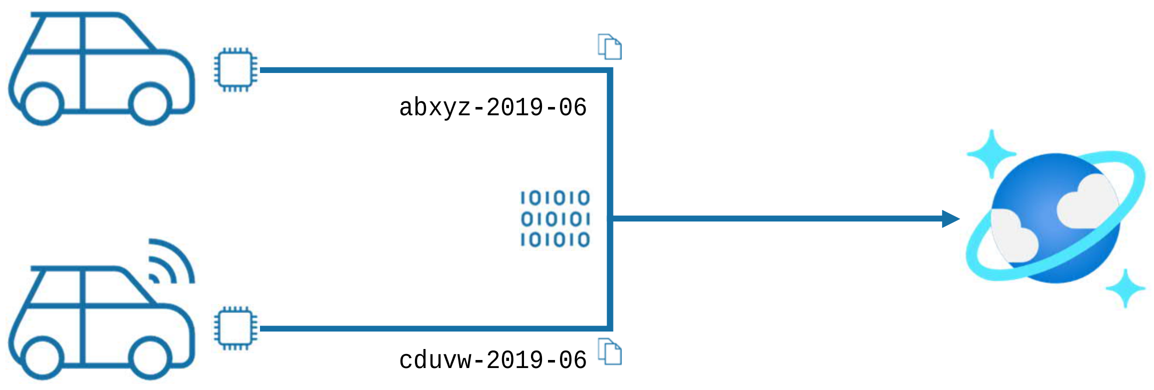
Microsoft

There's a problem with this assumption, as there are always outliers that will behave differently.
For example, consider a delivery driver who is paid by the hour, and spends six hours driving during an eight-hour shift.
They'll generate approximately 7 GBs of telemetry data per year.
This driver could reach the maximum storage amount for a logical partition within three years.

Therefore, to solve for the hourly driver scenario, we can add more granularity by creating a composite key, which concatenates the unique identifier for the vehicle with date information. For example, if the car has a unique id, such as a vehicle identification number (VIN) of "abxyz," we can create a composite partition key that is "abxyz-2019-06" to increase the granularity to car+year+month. Each month, the telemetry will use a new logical partition, which avoids this issue. This small additional amount of granularity safeguards against reaching that 20 GB maximum for a logical partition.
The key takeaway is that date alone is sometimes a poor choice for partition key because of the "hot partition" problem. Unique identifiers sometimes work but are susceptible to outliers maxing out their logical partition. Composite keys tend to work very well in practice.

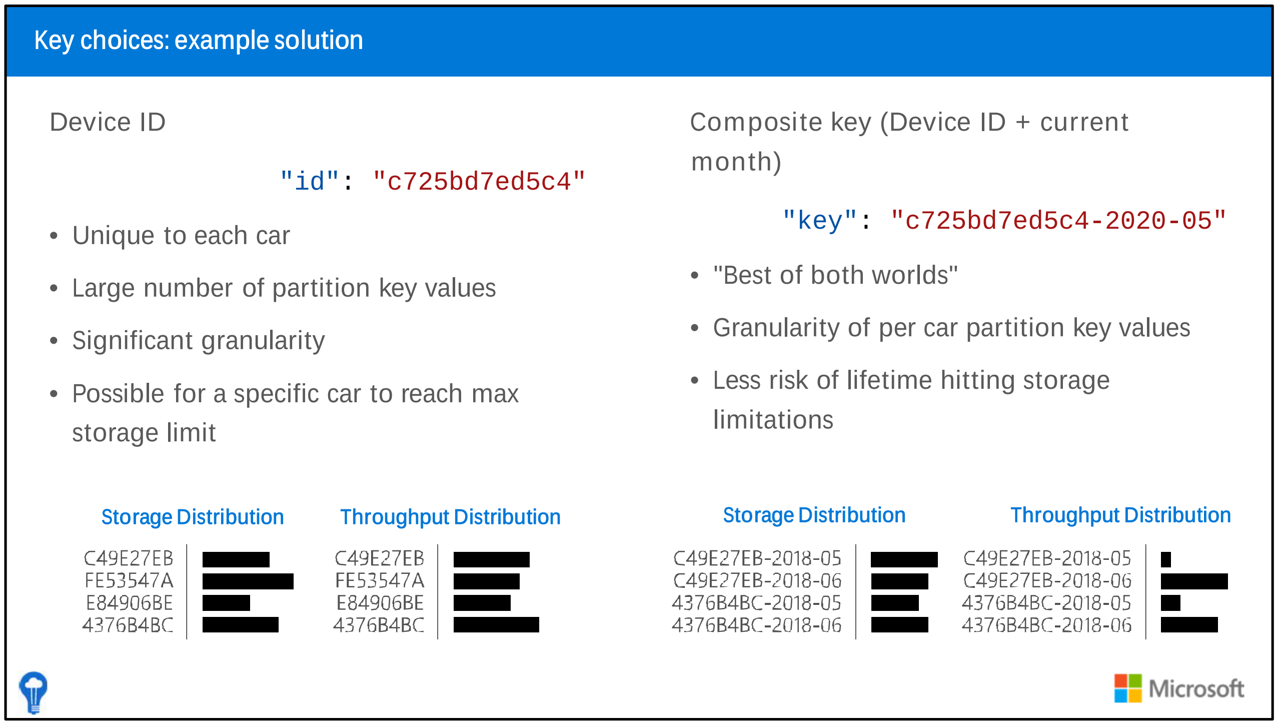


Device partition-key scenario: composite key



```
1 KB doc/sec * 360 minutes = ~21.09 MB/day
21.09 MB * 30.5 = ~0.02 GB/month
```

To solve for the hourly drive scenario, we can add more granularity by creating a composite key.
The composite key concatenates the unique identifier for the vehicle with some date information.
For example, if the car has a unique id (VIN) of “abxyz,” we can create a composite partition key that is “abxyz-2019-06” to increase the granularity to car+year+month.
Each month, the telemetry will use a new logical partition, avoiding this issue altogether.
This small additional amount of granularity safeguards us from reaching that 20 GB maximum for a logical partition.
The key takeaway is that date information alone is sometimes a poor choice for partition key because of the “hot partition” problem.
Unique identifiers sometimes work but are susceptible to outliers maxing out their logical partition. Composite keys tend to work very well in practice.



Key choices: example solution

Device ID

```
"id": "c725bd7ed5c4"
```

- Unique to each car
- Large number of partition key values
- Significant granularity
- Possible for a specific car to reach max storage limit

Composite key (Device ID + current month)

```
"key": "c725bd7ed5c4-2020-05"
```

- "Best of both worlds"
- Granularity of per car partition key values
- Less risk of lifetime hitting storage limitations

Storage Distribution

C49E27EB
FE53547A
E84906BE
4376B4BC

Throughput Distribution

C49E27EB
FE53547A
E84906BE
4376B4BC

Storage Distribution

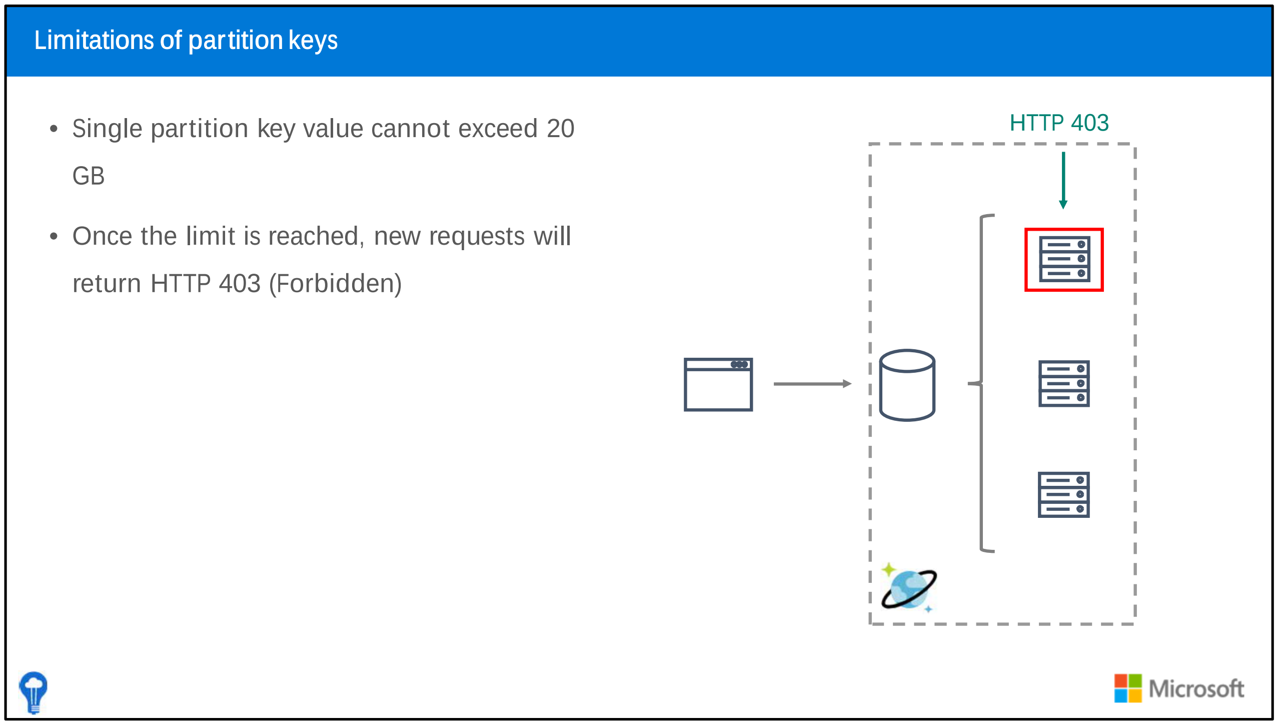
C49E27EB-2018-05
C49E27EB-2018-06
4376B4BC-2018-05
4376B4BC-2018-06

Throughput Distribution

C49E27EB-2018-05
C49E27EB-2018-06
4376B4BC-2018-05
4376B4BC-2018-06

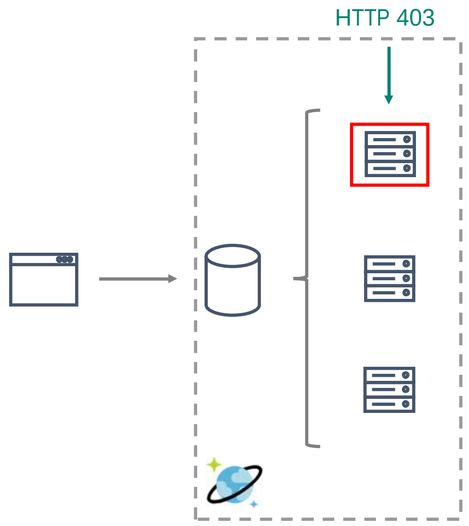
Microsoft

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 risk of reaching 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.



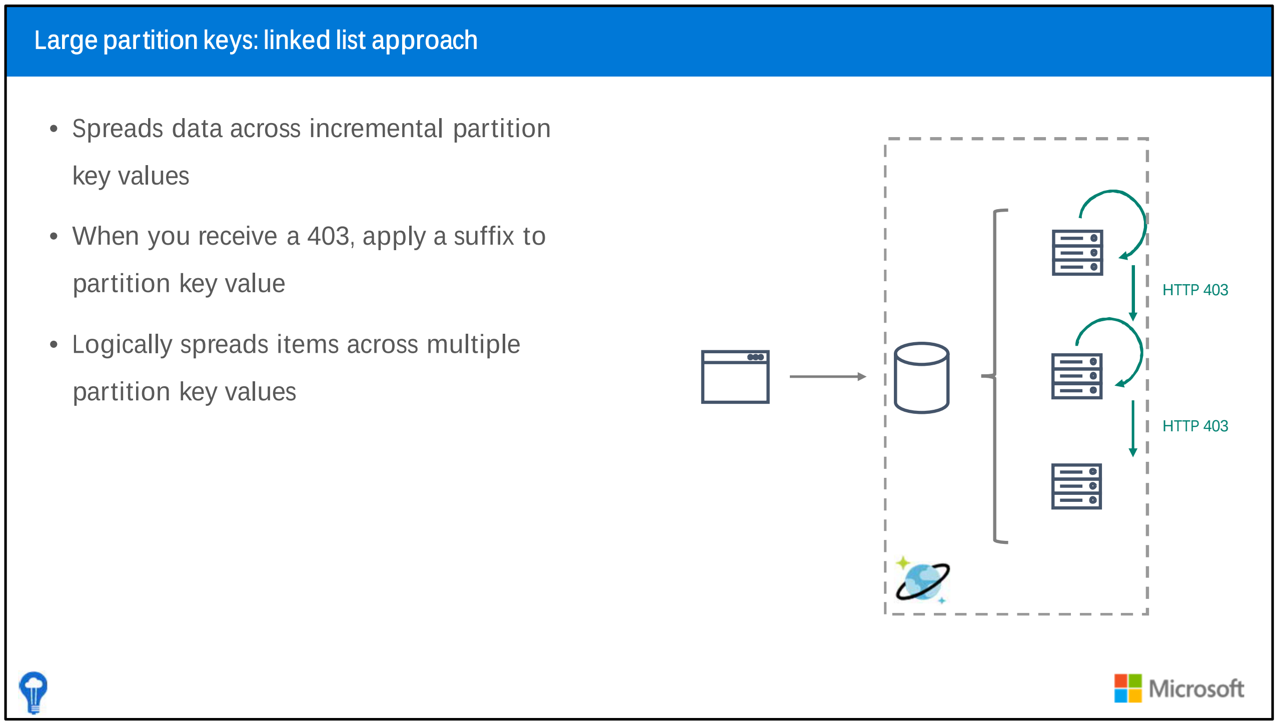
Limitations of partition keys

- Single partition key value cannot exceed 20 GB
- Once the limit is reached, new requests will return HTTP 403 (Forbidden)



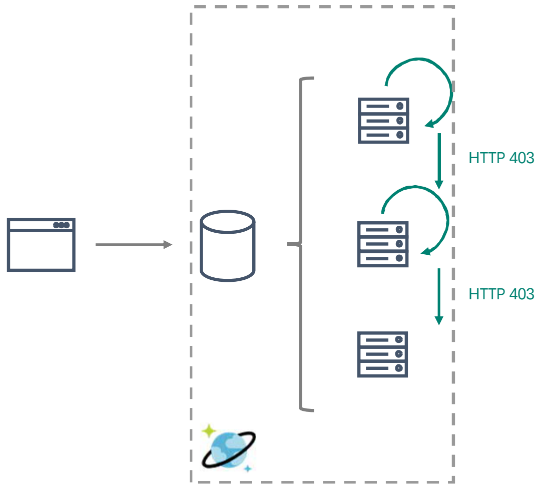
Microsoft

20 GB is the maximum for a single partition key value, which effectively makes it the maximum for a logical partition.
There are patterns that take advantage of the 403 error code.
You can endlessly increment partition key suffixes to work around any technical limitations for a specific partition (throughput and storage).



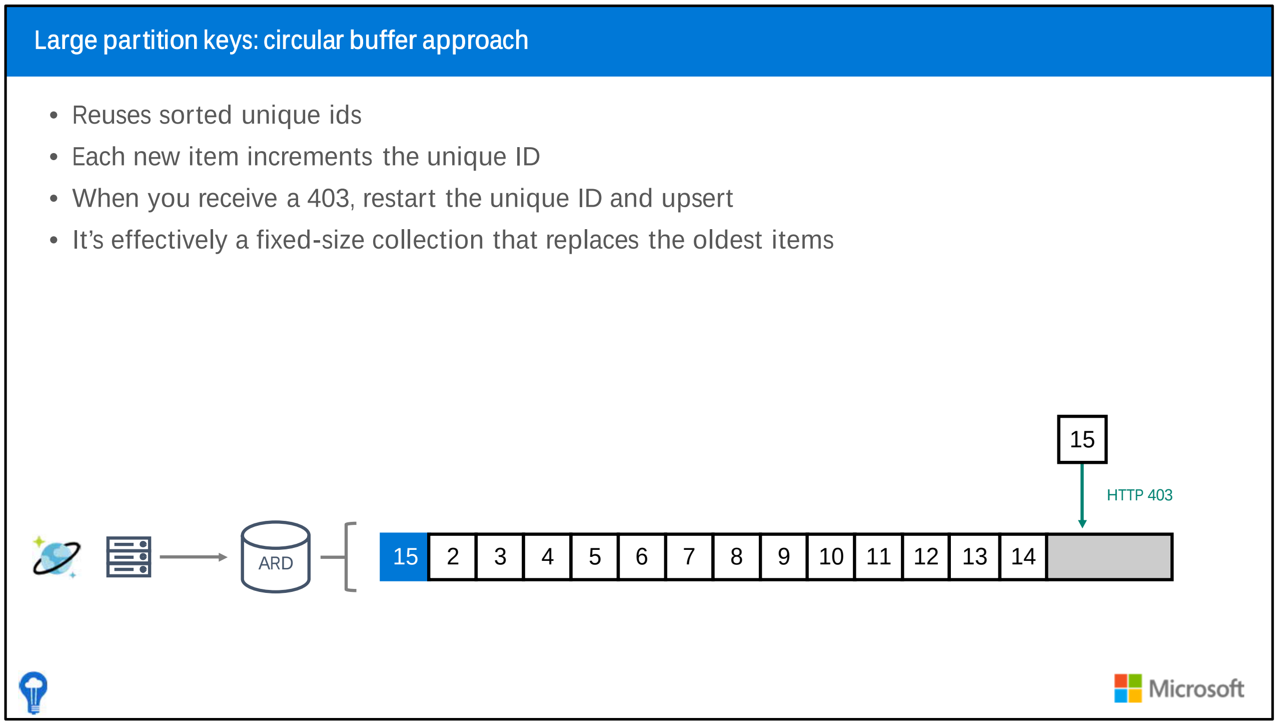
Large partition keys: linked list approach

- Spreads data across incremental partition key values
- When you receive a 403, apply a suffix to partition key value
- Logically spreads items across multiple partition key values



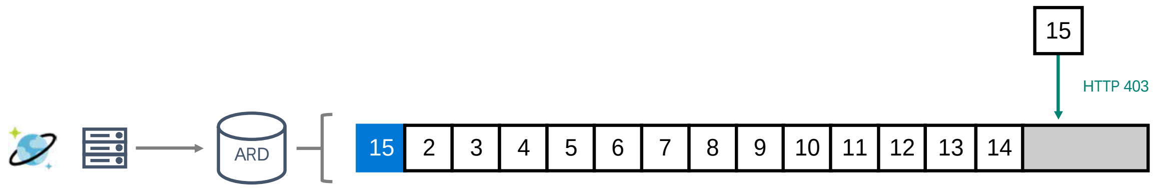
Microsoft

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 finding the 403 status code in your application's logic.

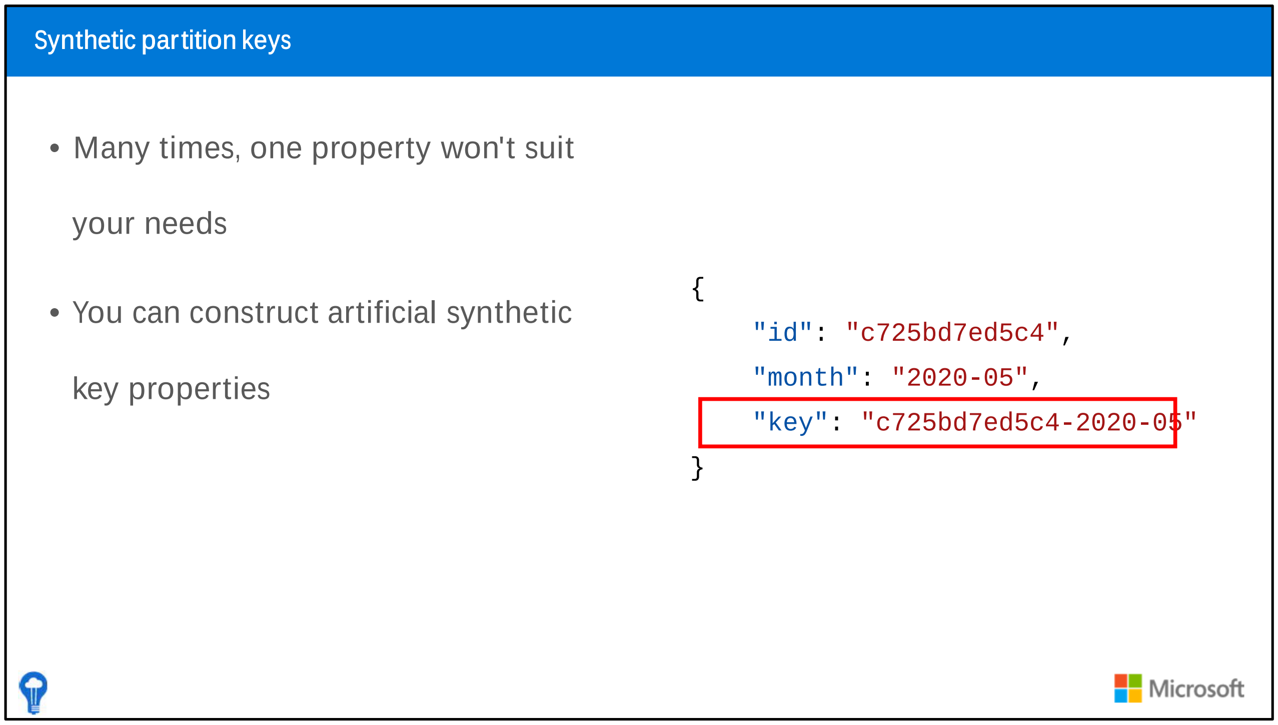


Large partition keys: circular buffer approach

- Reuses sorted unique ids
- Each new item increments the unique ID
- When you receive a 403, restart the unique ID and upsert
- It's effectively a fixed-size collection that replaces the oldest items



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.

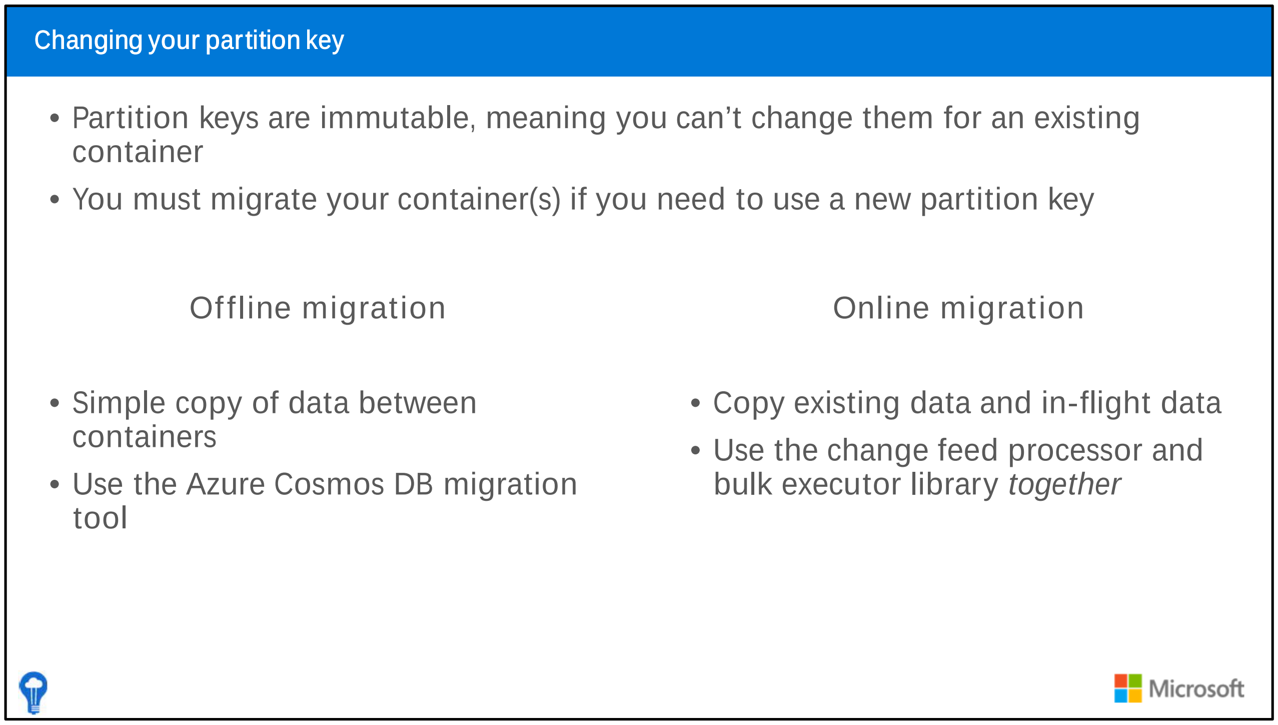


Synthetic partition keys

- Many times, one property won't suit your needs
- You can construct artificial synthetic key properties

```
{
    "id": "c725bd7ed5c4",
    "month": "2020-05",
    "key": "c725bd7ed5c4-2020-05"
}
```

https://aka.ms/Create_a_synthetic_partition_key



Changing your partition key

- Partition keys are immutable, meaning you can't change them for an existing container
- You must migrate your container(s) if you need to use a new partition key

Offline migration

- Simple copy of data between containers
- Use the Azure Cosmos DB migration tool

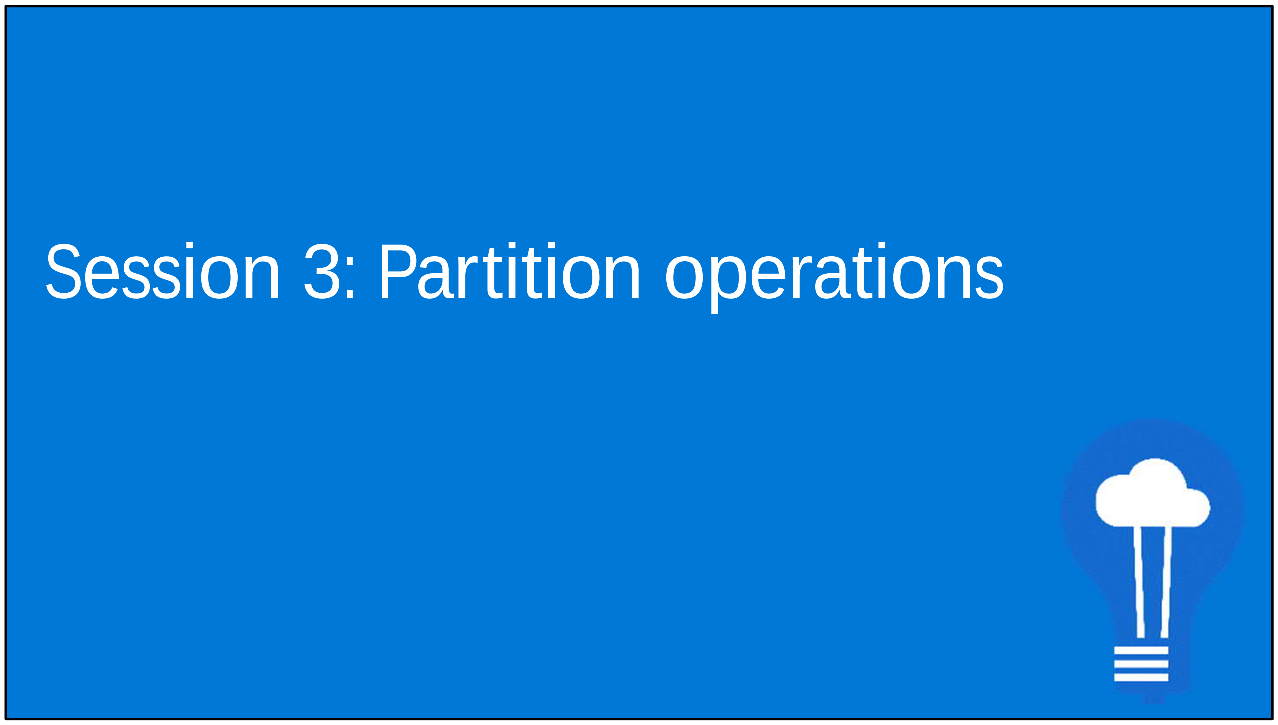
Online migration

- Copy existing data and in-flight data
- Use the change feed processor and bulk executor library *together*

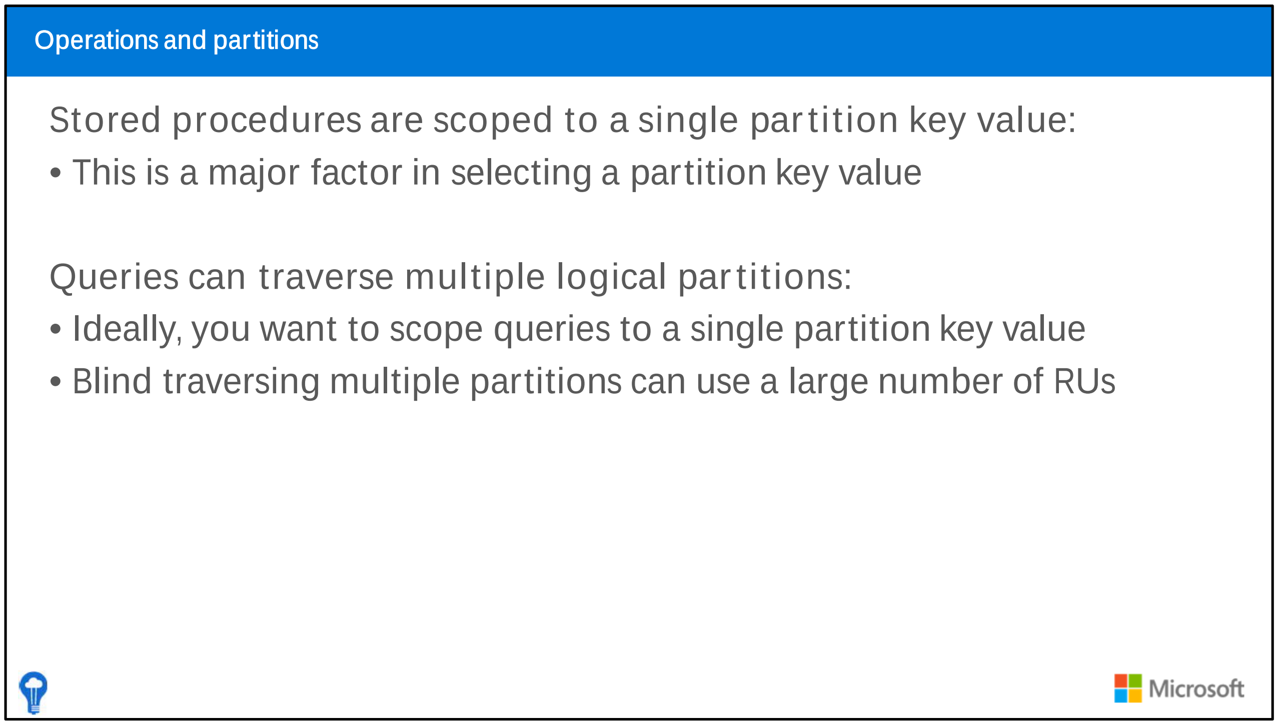
Microsoft

https://aka.ms/Data_migration_tool
https://aka.ms/Change_the_feed_processor_in_Azure_Cosmos_DB
https://aka.ms/Azure_Cosmos_DB_bulk_executor_library_overview
Optionally, you can write custom code to perform a migration.
The product team has already built a live application migrator utility:
https://aka.ms/Azure_Cosmos_db_live_data_migrator
Live migration is also useful when:

    You wish to change your database/container throughput provisioning scheme.
    A team member needs to copy existing data to troubleshoot in a nonproduction environment.
    Renaming a container.



Session 3: Partition operations



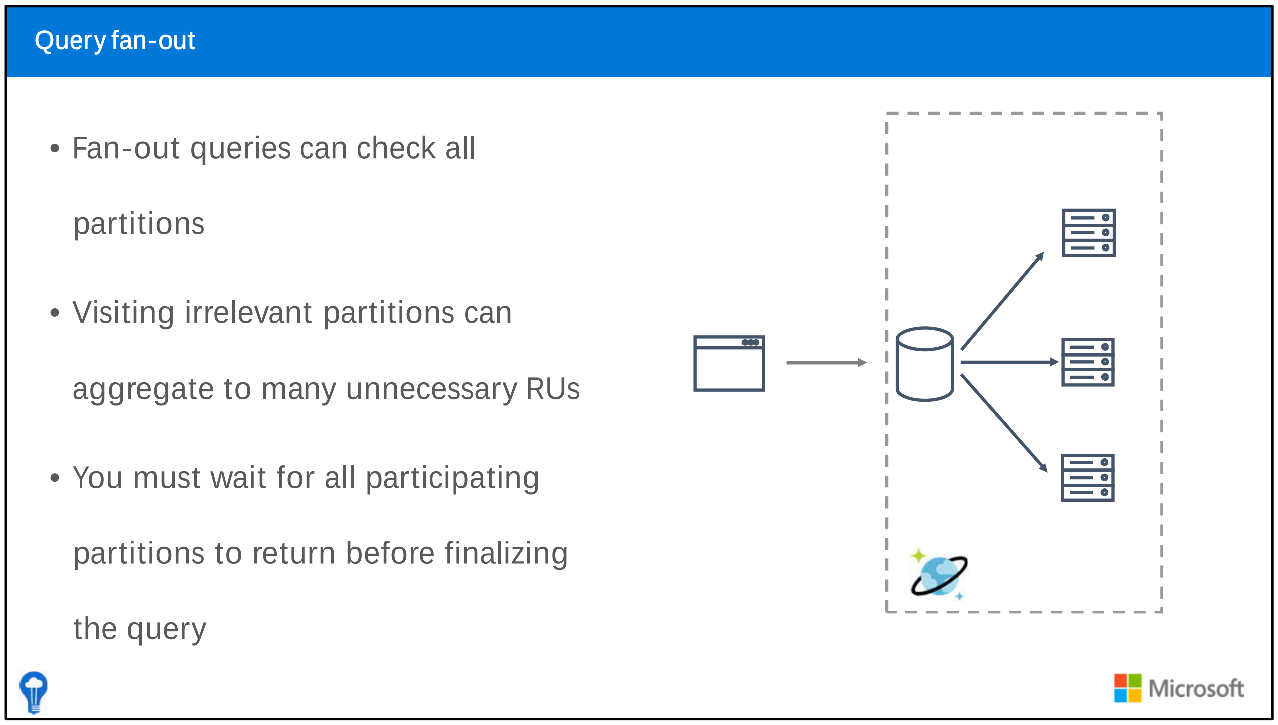
Operations and partitions

Stored procedures are scoped to a single partition key value:
- This is a major factor in selecting a partition key value

Queries can traverse multiple logical partitions:
- Ideally, you want to scope queries to a single partition key value
- Blind traversing multiple partitions can use a large number of RUs

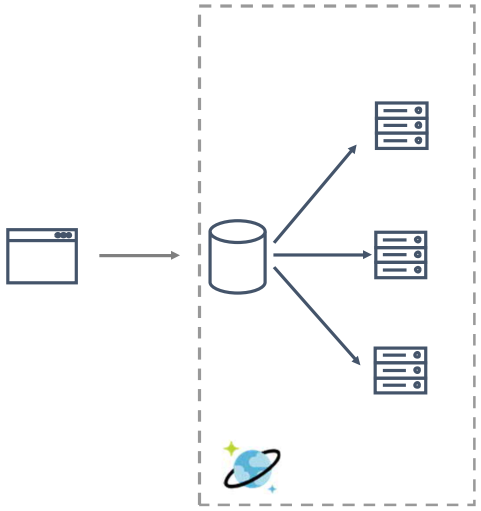
Microsoft

If a query has a filter that matches your partition key, only relevant partition's indexes are checked. This optimizes the RUs used.
https://aka.ms/Troubleshoot_query_issues_when_using_Azure_Cosmos_DB
You are charged approximately one RU for each partition you visit that doesn't have relevant data.
Multiple fan-out queries will max out RU per seconds for a partition.



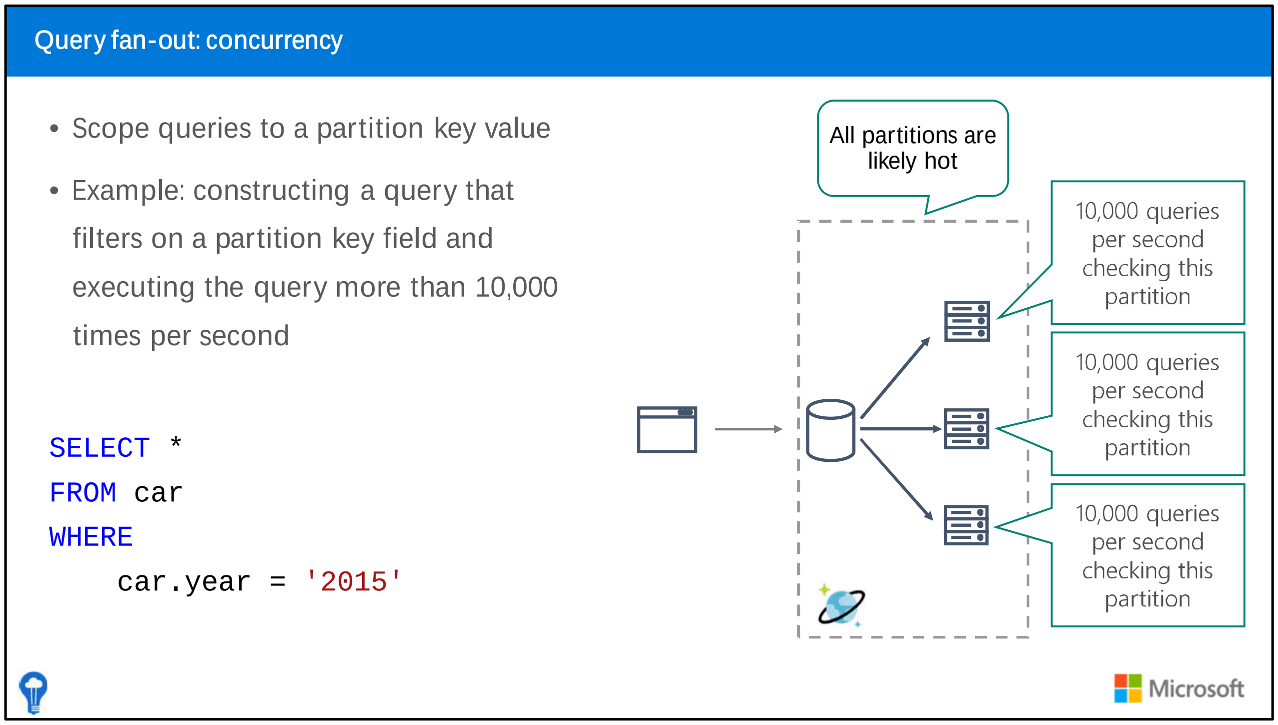
Query fan-out

- Fan-out queries can check all partitions
- Visiting irrelevant partitions can aggregate to many unnecessary RUs
- You must wait for all participating partitions to return before finalizing the query



Microsoft

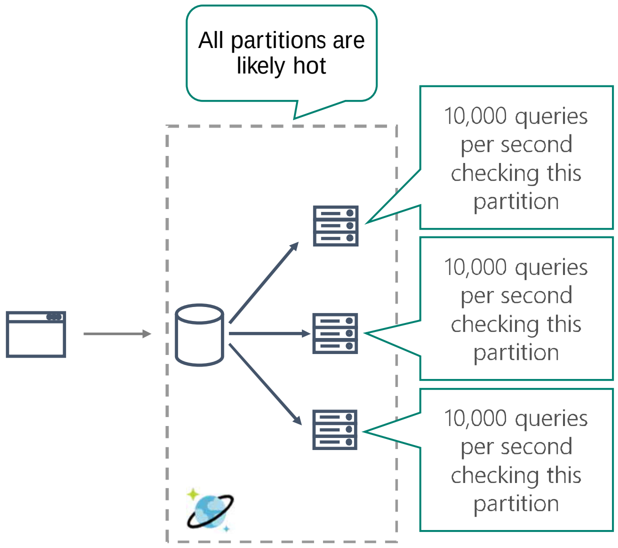
Fan-out of queries aren’t necessarily bad, as you can tune them and make them parallel.



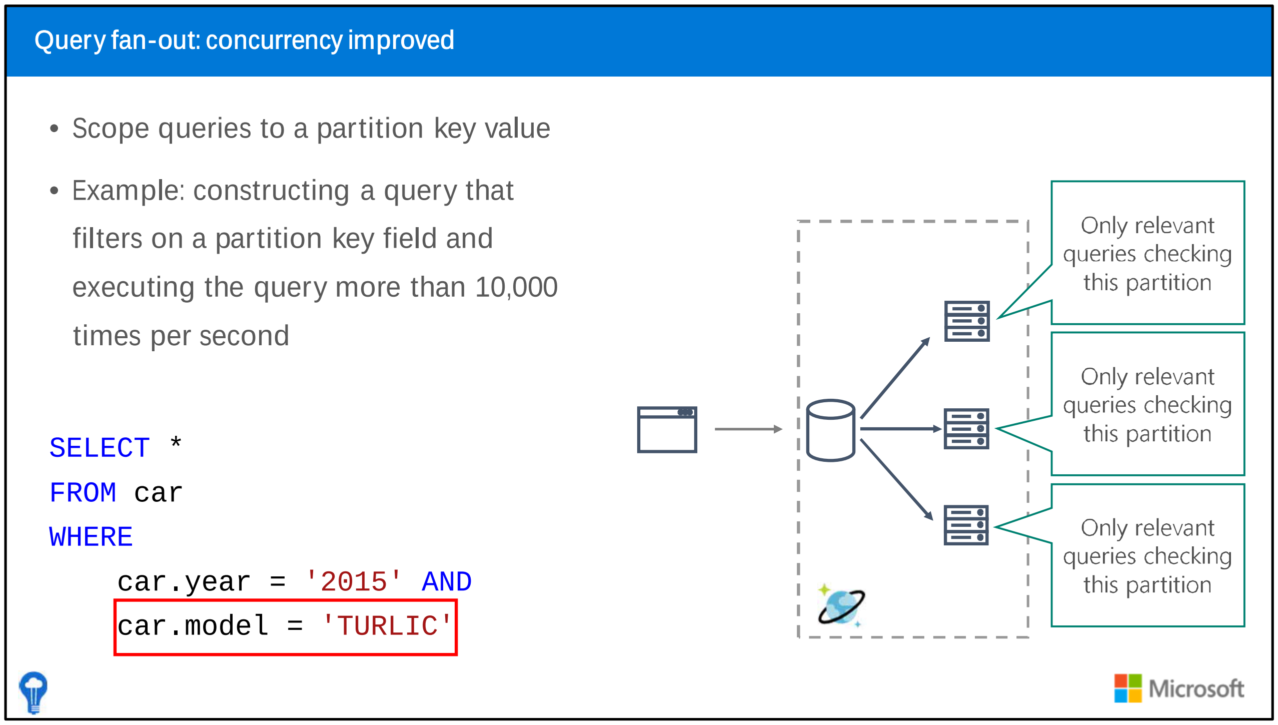
Query fan-out: concurrency

- Scope queries to a partition key value
- Example: constructing a query that filters on a partition key field and executing the query more than 10,000 times per second

```
SELECT *
FROM car
WHERE
    car.year = '2015'
```


All partitions are likely hot
10,000 queries per second checking this partition
10,000 queries per second checking this partition
10,000 queries per second checking this partition

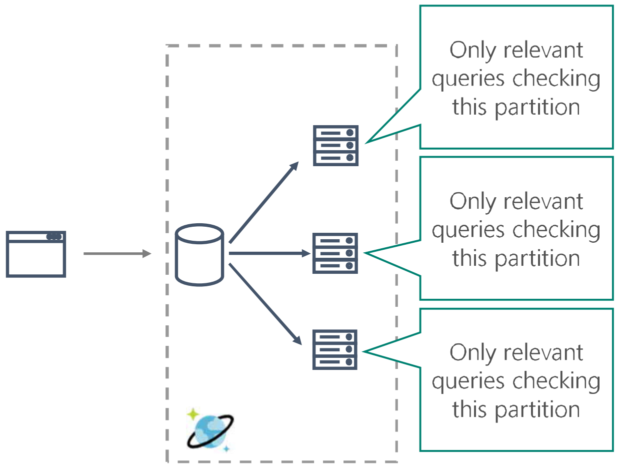
Fan-out queries can quickly grow and become too large if executed multiple times.
You should only perform fan-out queries in strategic scenarios where you understand the implications.



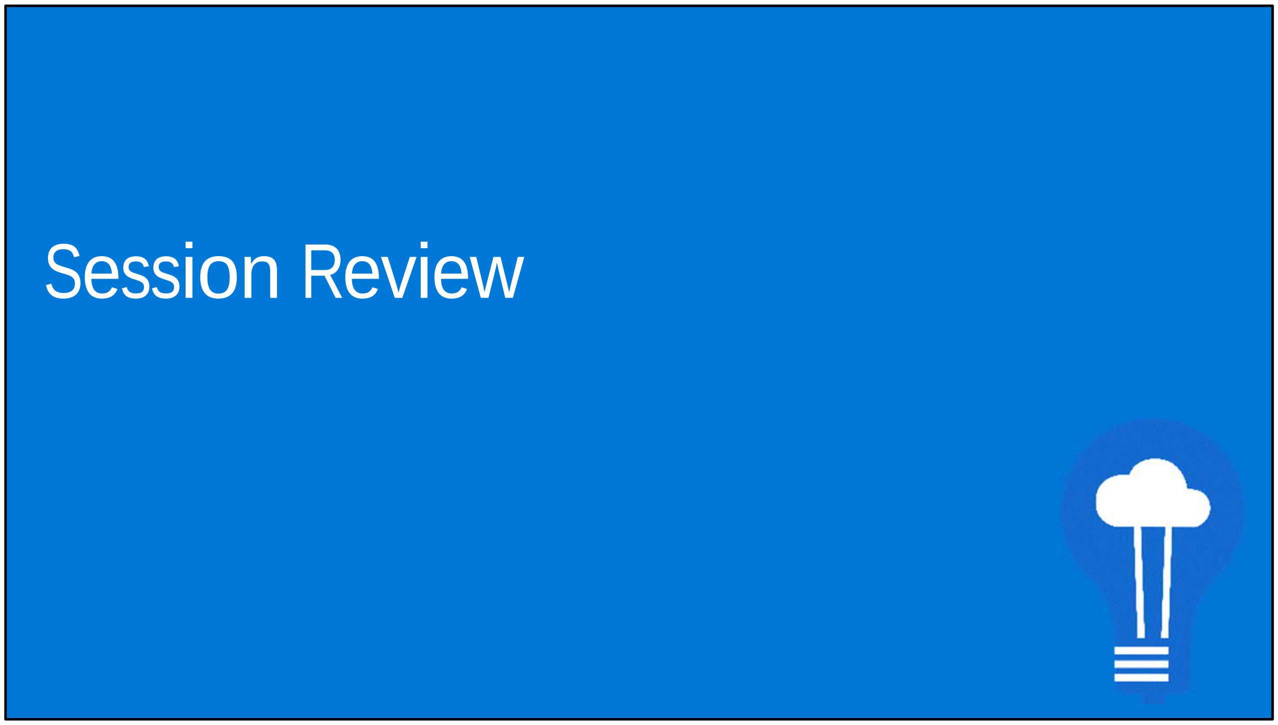
Query fan-out: concurrency improved

- Scope queries to a partition key value
- Example: constructing a query that filters on a partition key field and executing the query more than 10,000 times per second

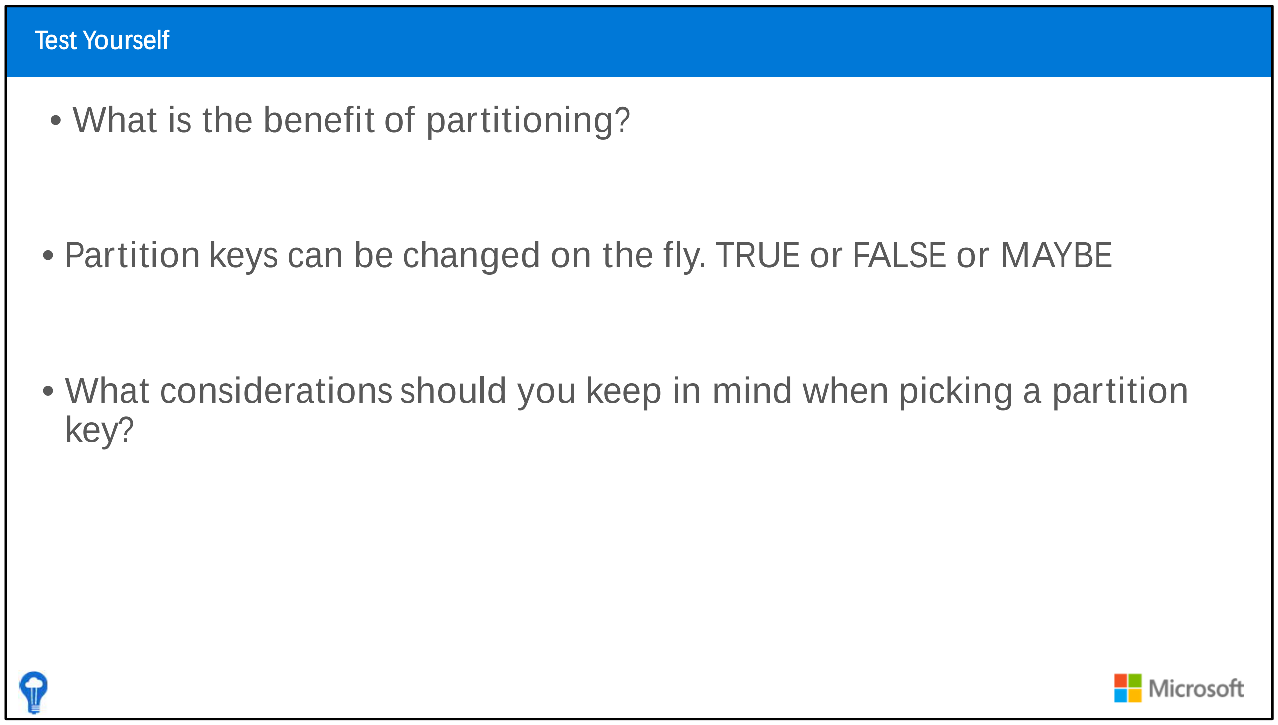
```
SELECT *
FROM car
WHERE
    car.year = '2015' AND
    car.model = 'TURLIC'
```


Only relevant queries checking this partition
Only relevant queries checking this partition
Only relevant queries checking this partition

In this example, model is the partition key.
Fan-out of queries are not always bad, as you can tune them and make them parallel.
Ideally, queries are filtered to include relevant partitions only.



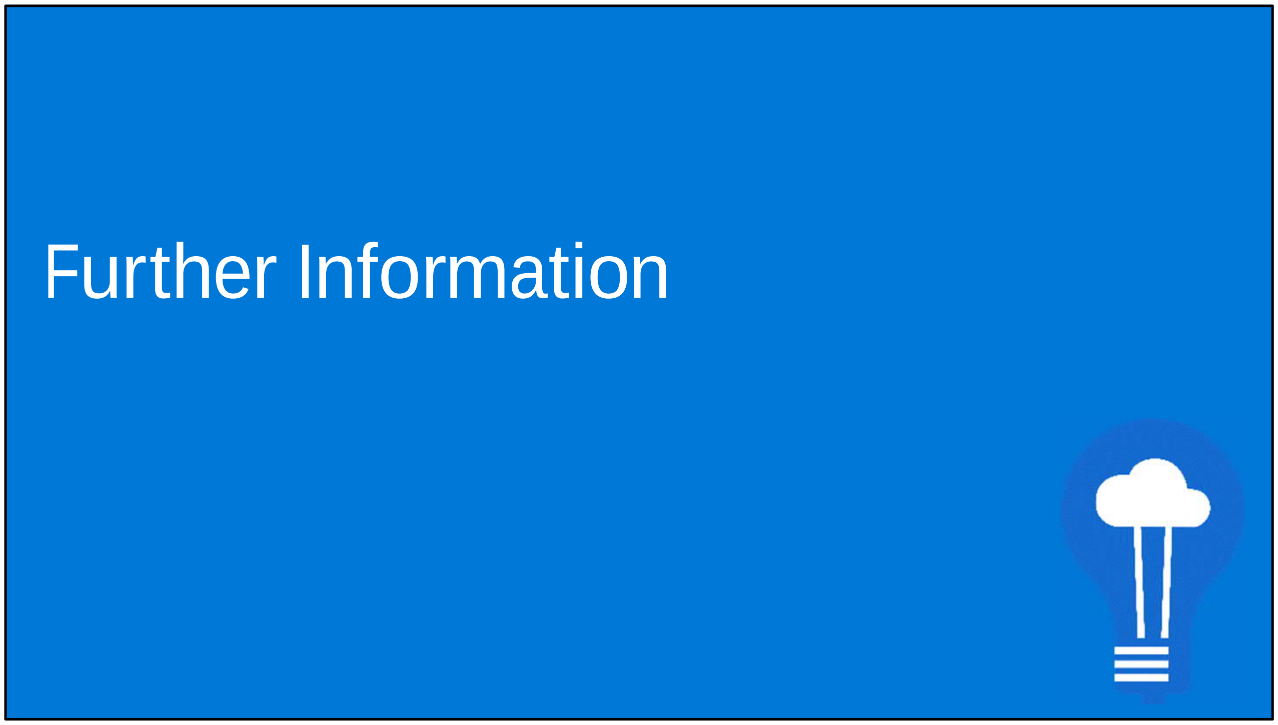
Session Review



Test Yourself

- What is the benefit of partitioning?
- Partition keys can be changed on the fly. TRUE or FALSE or MAYBE
- What considerations should you keep in mind when picking a partition key?

Microsoft



Further Information

References

- https://docs.microsoft.com/en-us/azure/cosmos-db/
- https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview
- https://docs.microsoft.com/en-us/azure/cosmos-db/change-feed

Microsoft