Microsoft

Implement a data modeling and partitioning strategy for Azure Cosmos DB for NoSQL

Agenda

- Implement a non-relational data model
- Design a data partitioning strategy

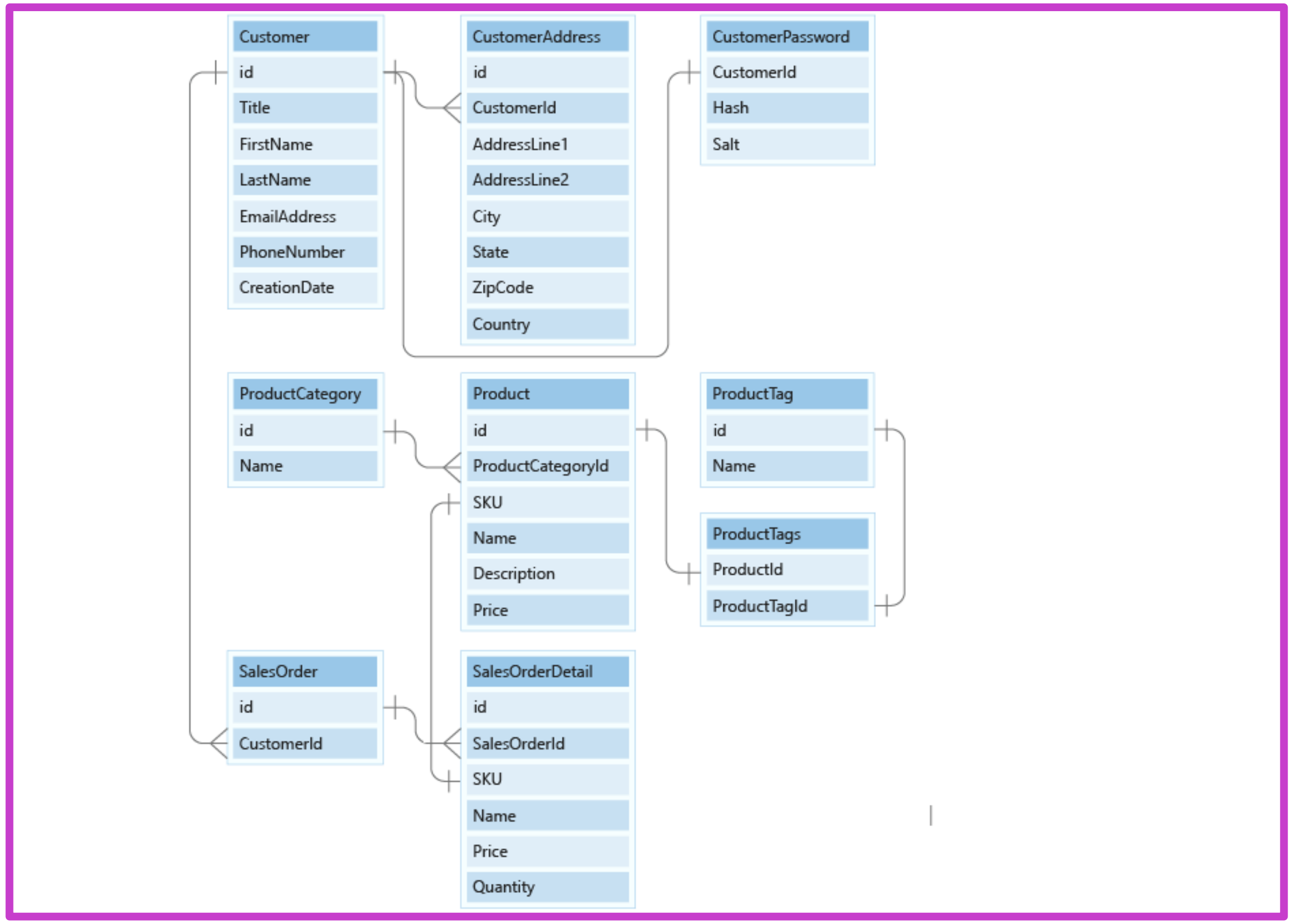
Implement a non-relational data model

What’s the difference between NoSQL and relational databases

Relational DB	Vertical Scaling	Defined Schema	Relational
NoSQL	Horizontal Scaling	No predefined Schema	Non-Relational

Scenario

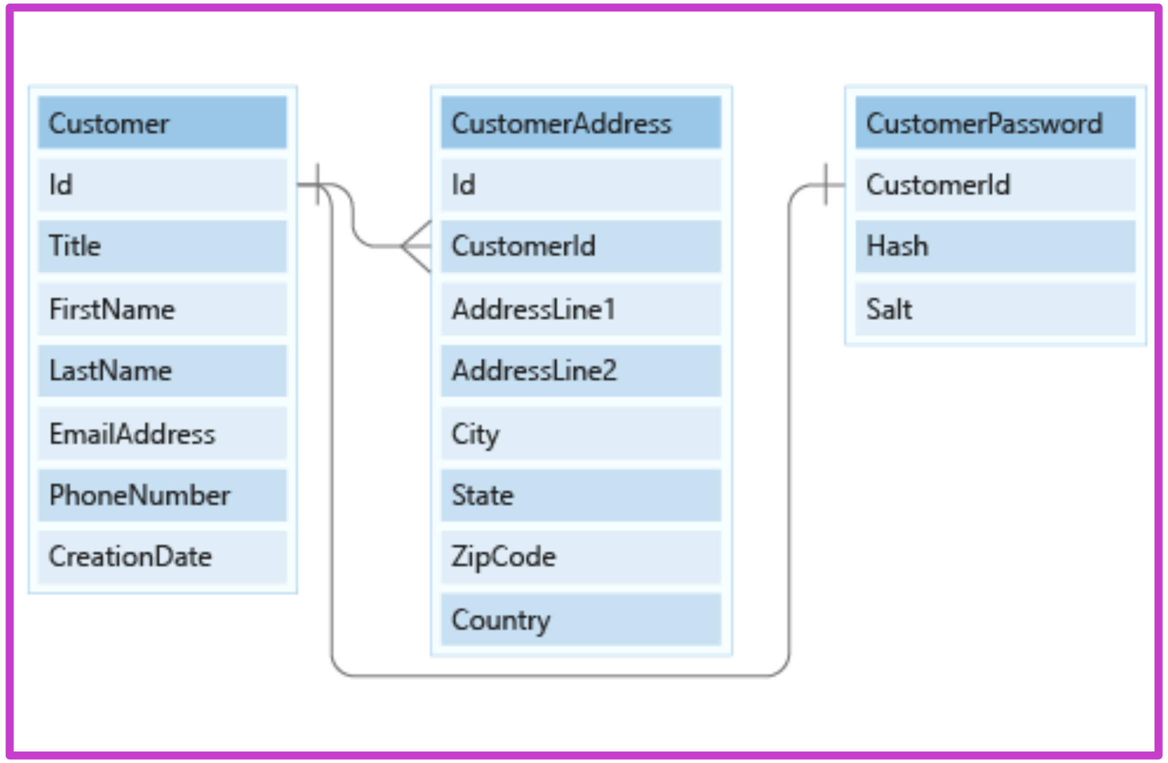
The following entity-relationship (ER) diagram details the nine entities used in the next sections.



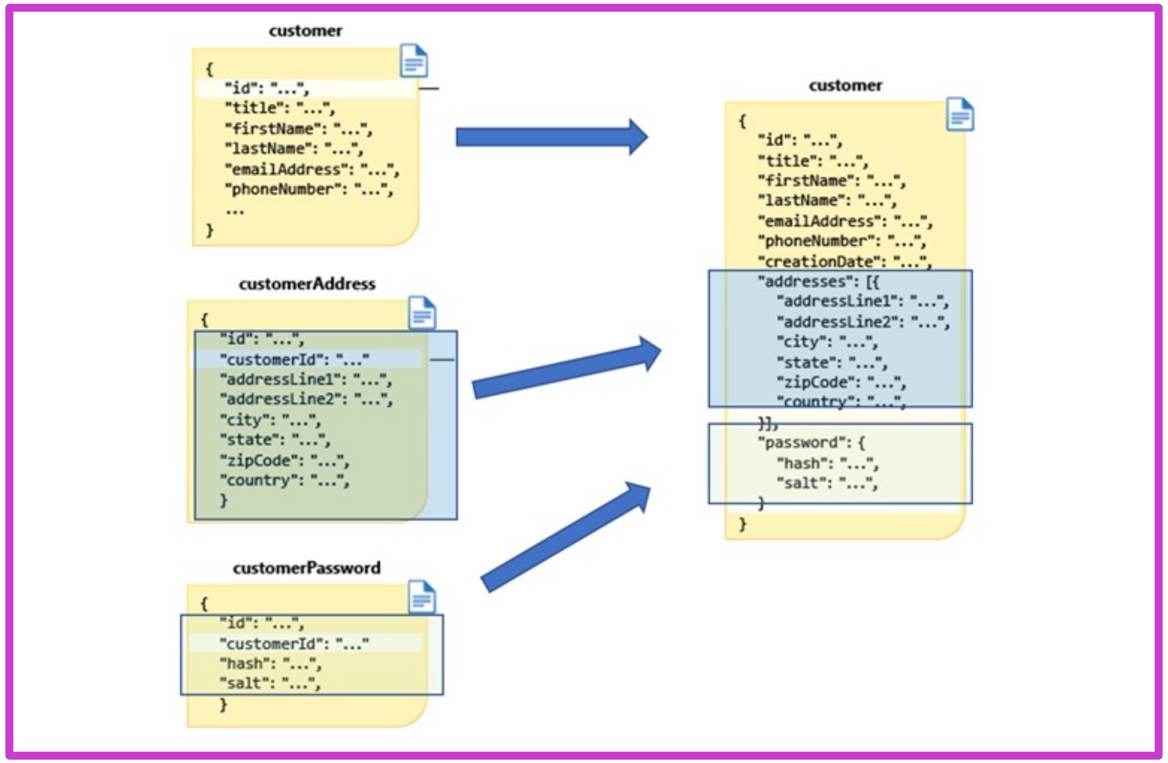
Identify access patterns for your app

When you’re designing a data model for a NoSQL database, the objective is to ensure that operations on data are done in the fewest requests.

Identify access patterns for customer entities



Model customer entities



```
{
  "id": "...",
  "title": "...",
  "firstName": "...",
  "lastName": "...",
  "emailAddress": "...",
  "phoneNumber": "...",
  ...
}
```

```
{
  "id": "...",
  "customerId": "..."
  "addressLine1": "...",
  "addressLine2": "...",
  "city": "...",
  "state": "...",
  "zipCode": "...",
  "country": "...",
  }
```

```
{
  "id": "...",
  "customerId": "..."
  "hash": "...",
  "salt": "...",
  }
```

```
{
  "id": "...",
  "title": "...",
  "firstName": "...",
  "lastName": "...",
  "emailAddress": "...",
  "phoneNumber": "...",
  "creationDate": "...",
  "addresses": [{
    "addressLine1": "...",
    "addressLine2": "...",
    "city": "...",
    "state": "...",
    "zipCode": "...",
    "country": "...",
  }],
  "password": {
    "hash": "...",
    "salt": "...",
  }
}
```

When to embed or reference data

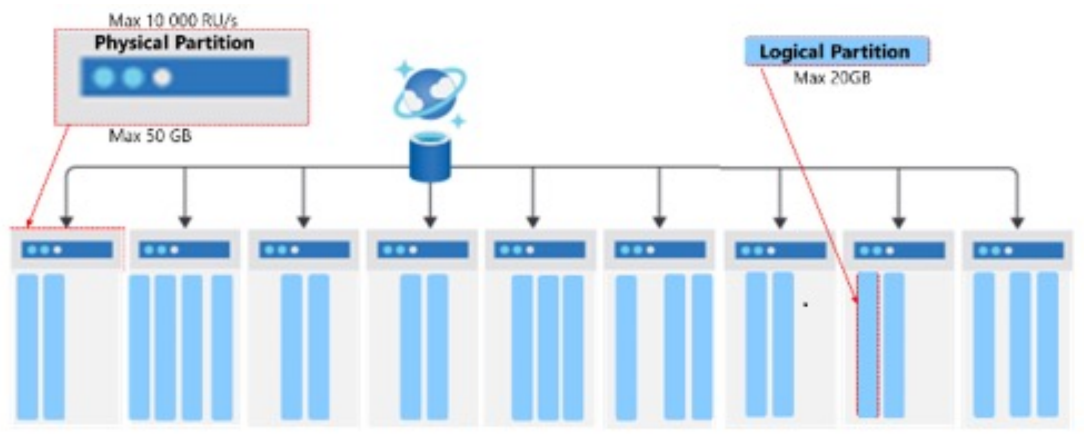
There are rules for when you should embed data in a document instead of referencing it in a different row.

When should you embed data?	Read or updated together 1:1 relationship 1: Few relationship
When should you reference data?	Read or updated independently 1: Many relationship Many: Many relationship

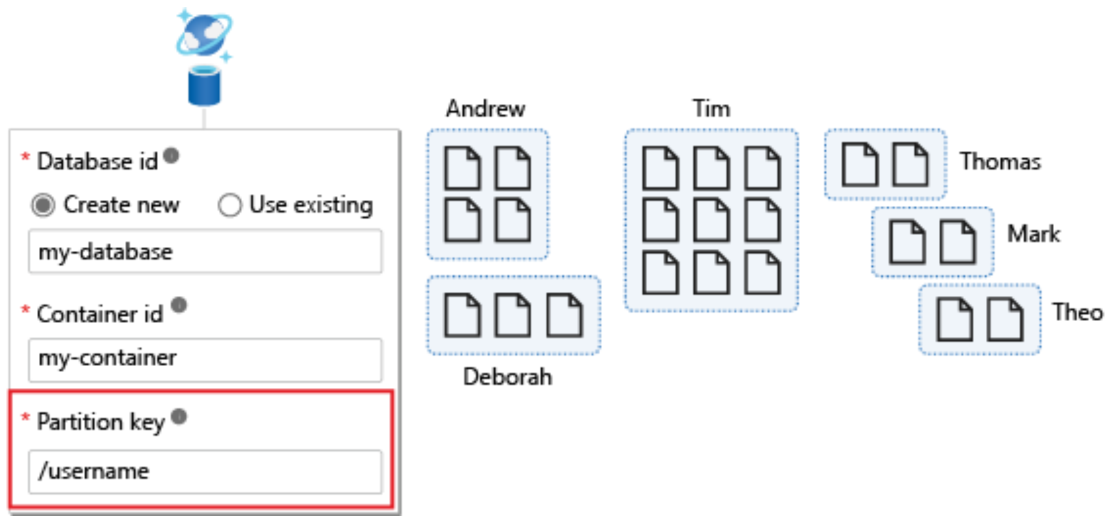
Choose a partition key

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.

Physical and Logical partitions in Azure Cosmos DB



Partitions Key

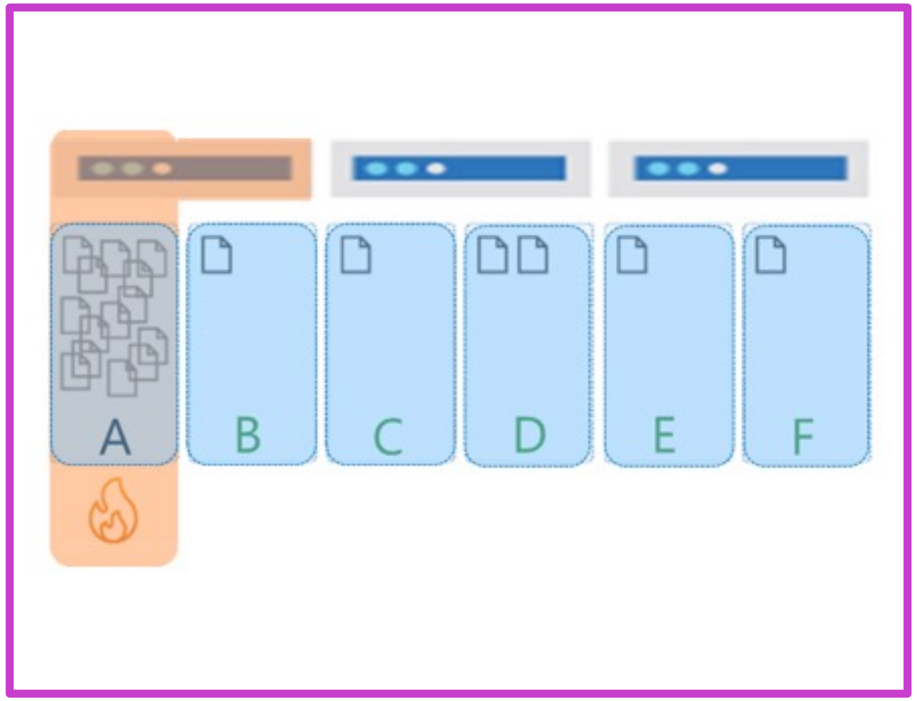


* Database id	[✓ Create new] Use existing
	my-database
* Container id	my-container
* Partition key	/username

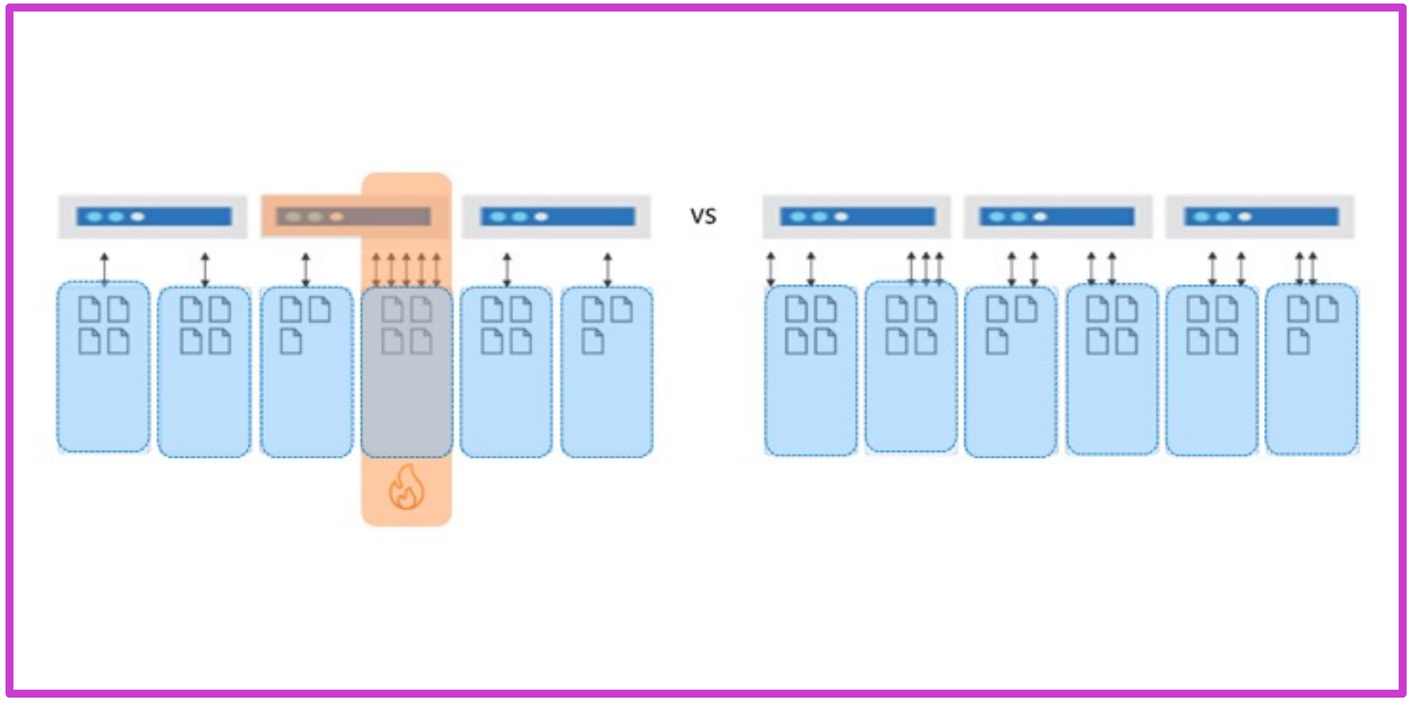
Avoid hot partitions

When data is not partitioned correctly, it can result in *hot partitions*. Hot partitions prevent your application workload from scaling.

Storage hot partitions



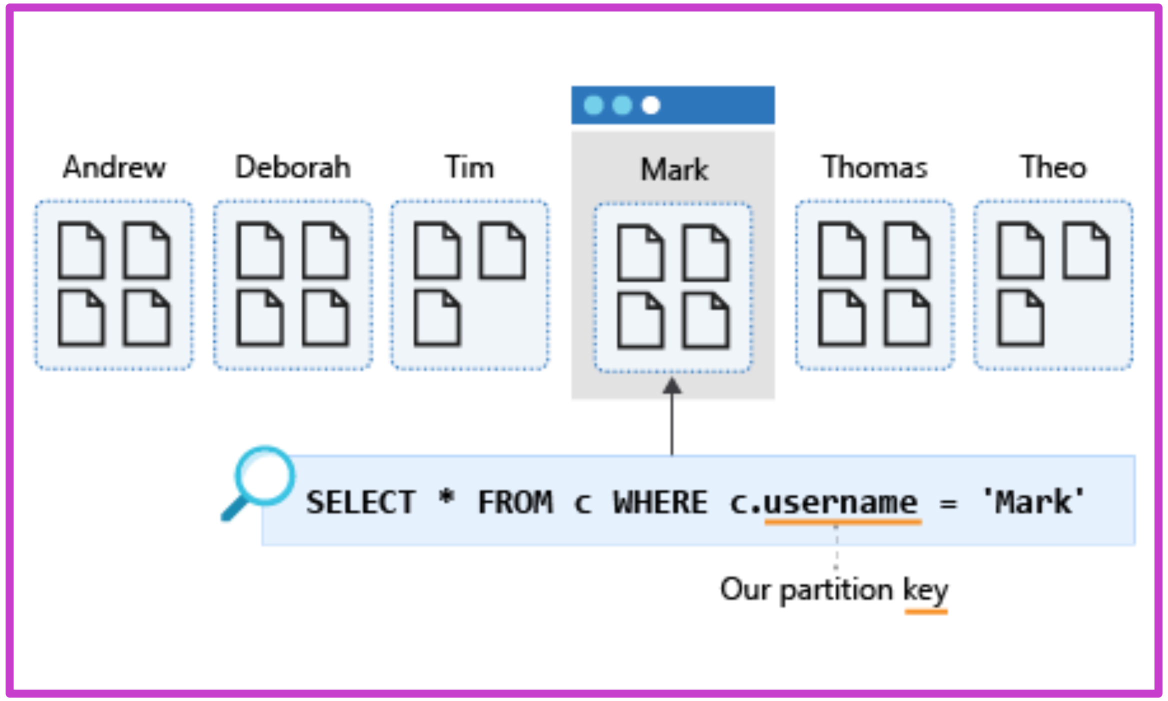
Throughput hot partitions



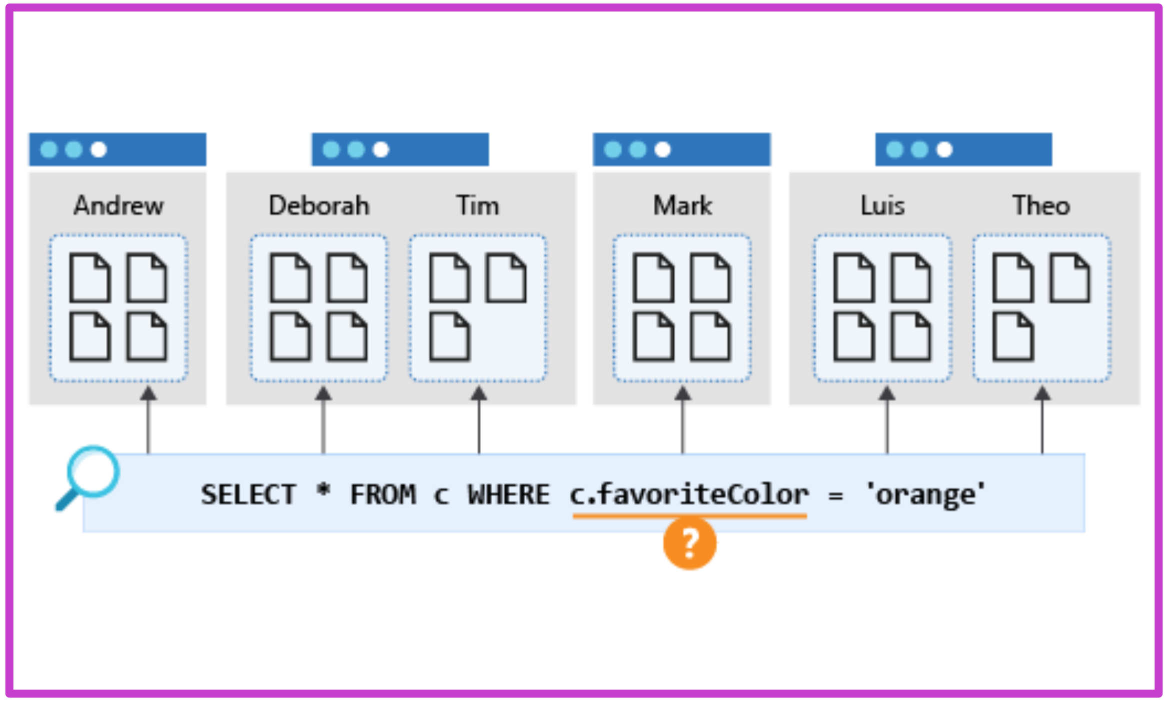
Consider reads versus writes

When you're choosing a partition key, you also need to consider whether the data is read heavy or write heavy.

Single Partition Query

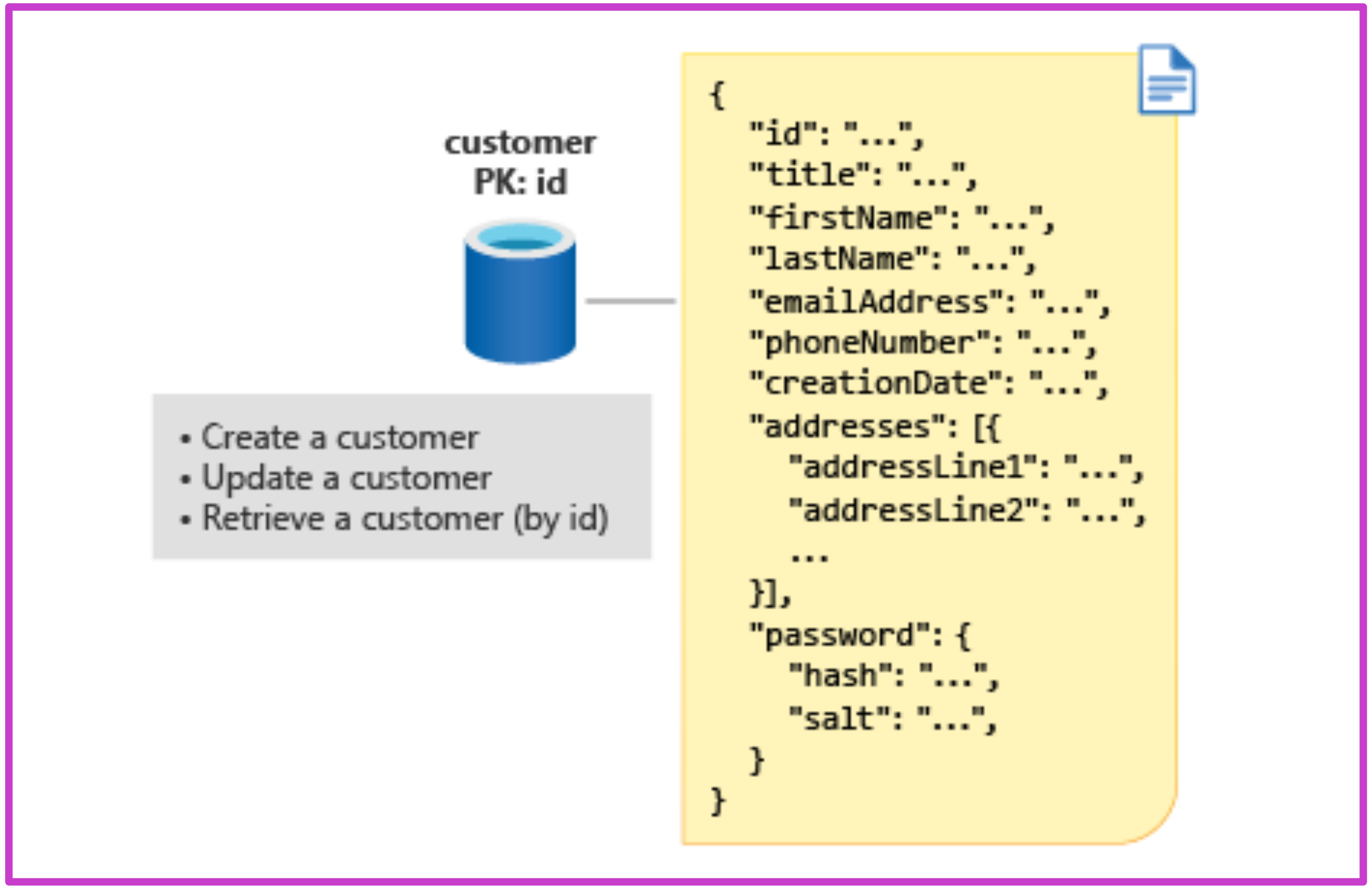


Cross Partition Query



Choose a partition key for customers

Now that you understand partitioning in Azure Cosmos DB, we can decide on a partition key for our customer data.



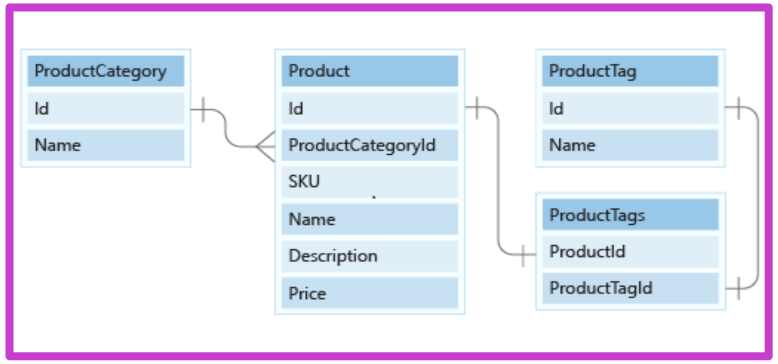
- Create a customer
- Update a customer
- Retrieve a customer (by id)

```
{
  "id": "...",
  "title": "...",
  "firstName": "...",
  "lastName": "...",
  "emailAddress": "...",
  "phoneNumber": "...",
  "creationDate": "...",
  "addresses": [{
    "addressLine1": "...",
    "addressLine2": "...",
    ...
  }],
  "password": {
    "hash": "...",
    "salt": "...",
  }
}
```

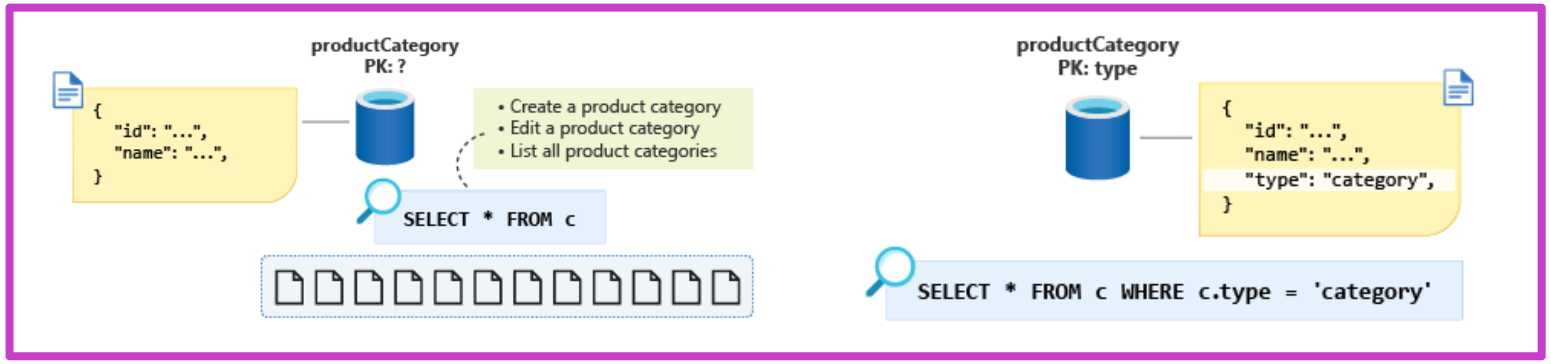
Model small lookup entities

Our data model includes two small reference data entities, *ProductCategory* and *ProductTag*.

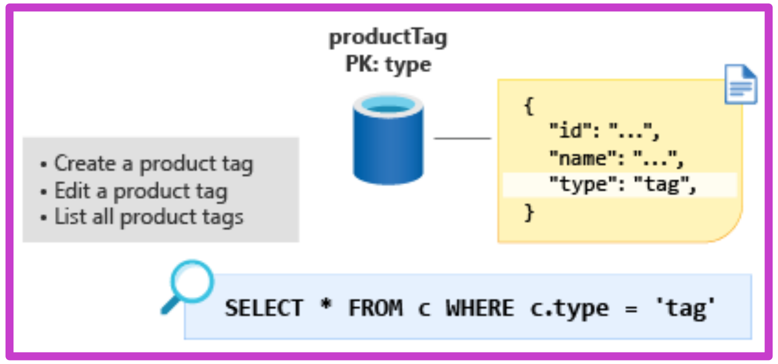
Product relational model



Model product categories



Model product tags



Lab – Measure performance for customer entities

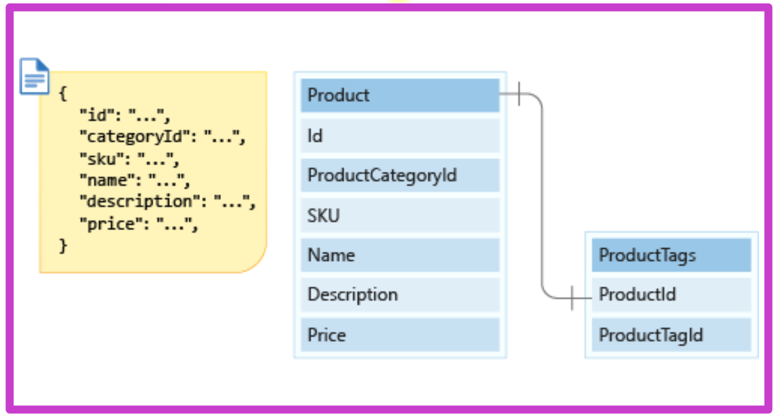
- Prepare your development environment
- Measure performance of entities in separate containers
- Measure performance of embedded entities
- Compare the performance of the two models

Design a data partitioning strategy

Denormalize data in your model

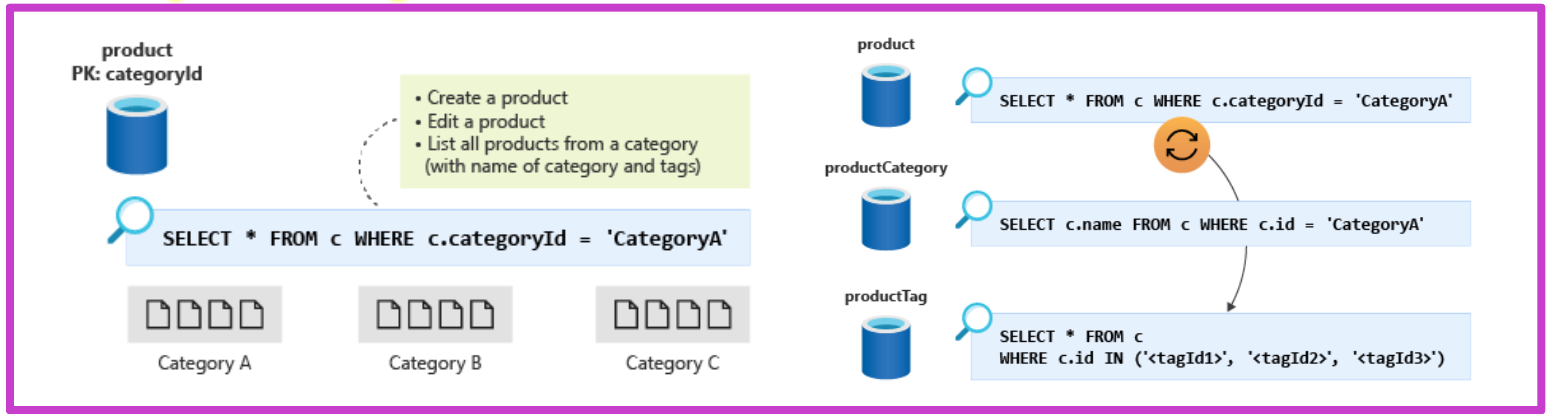
Let’s model a *product* table from your relational database into a NoSQL database.

Product/Product Tags relational model



```
{
  "id": "...",
  "categoryId": "...",
  "sku": "...",
  "name": "...",
  "description": "...",
  "price": "...",
}
```

Select a partition key



- Create a product
- Edit a product
- List all products from a category (with name of category and tags)

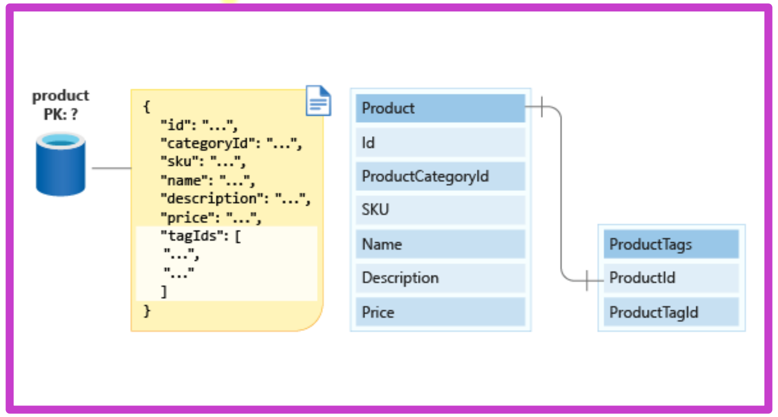
```
SELECT * FROM c WHERE c.categoryId = 'CategoryA'
```

```
SELECT * FROM c WHERE c.categoryId = 'CategoryA'
```

```
SELECT c.name FROM c WHERE c.id = 'CategoryA'
```

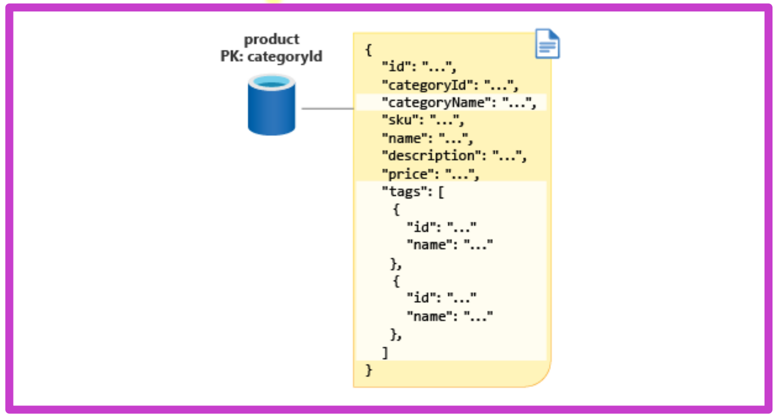
```
SELECT * FROM c
WHERE c.id IN ('<tagId1>', '<tagId2>', '<tagId3>')
```

Model the product entities



```
{
  "id": "...",
  "categoryId": "...",
  "sku": "...",
  "name": "...",
  "description": "...",
  "price": "...",
  "tagIds": [
    "...",
    "..."
  ]
}
```

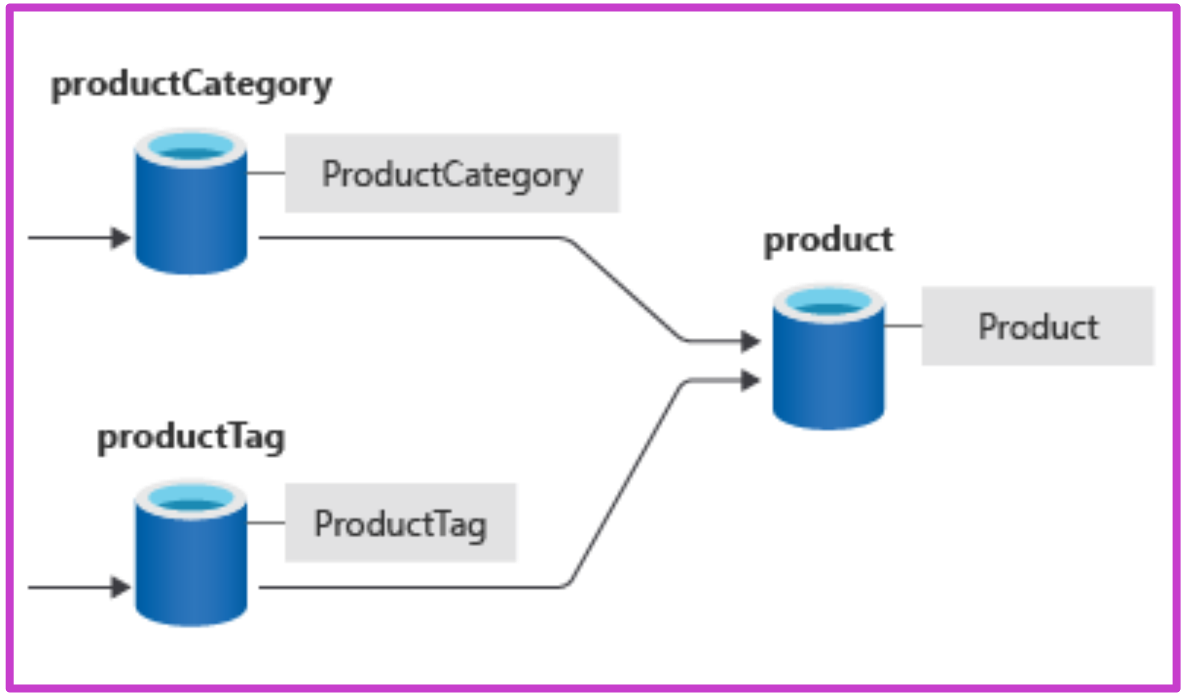
Denormalize product entities



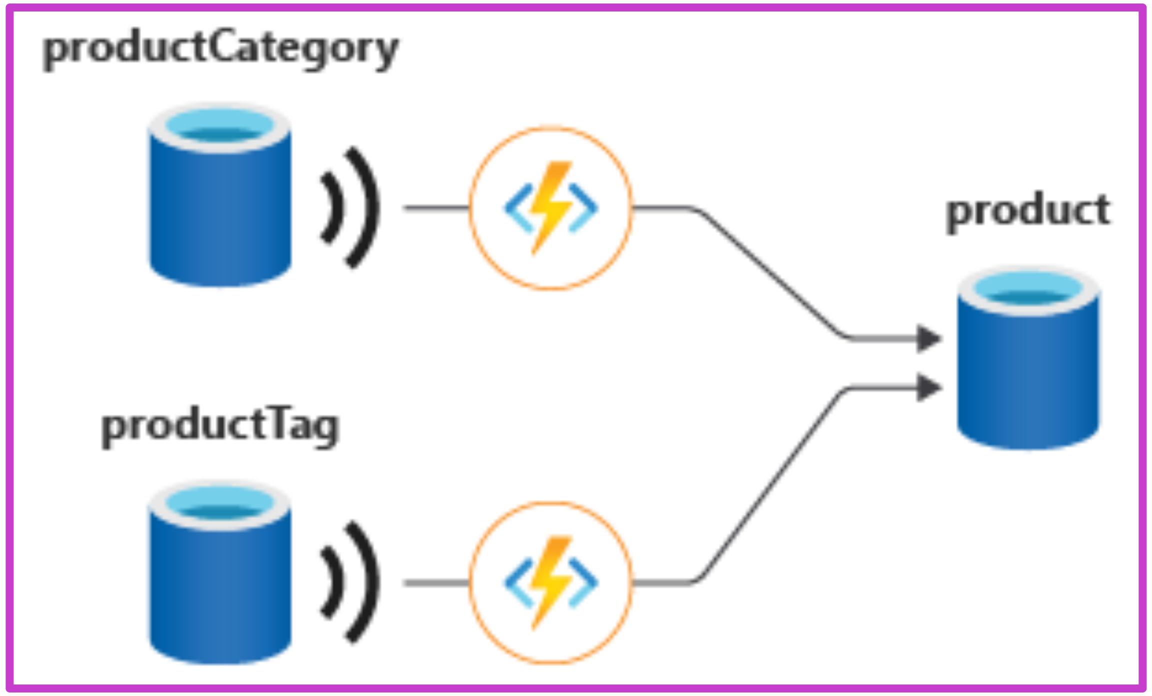
```
{
  "id": "...",
  "categoryId": "...",
  "categoryName": "...",
  "sku": "...",
  "name": "...",
  "description": "...",
  "price": "...",
  "tags": [
    {
      "id": "..."
      "name": "..."
    },
    {
      "id": "..."
      "name": "..."
    },
  ]
}
```

Manage referential integrity by using change feed

When denormalizing data like shown for the Product, ProductCategory and ProductTag tables, referential integrity must be maintained when changes occur on the categories or tags.



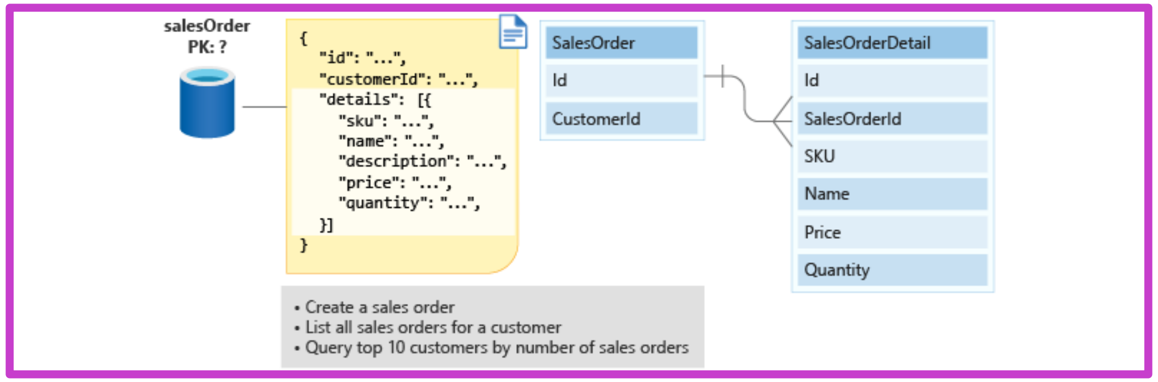
Change Feed



Combine multiple entities in the same container

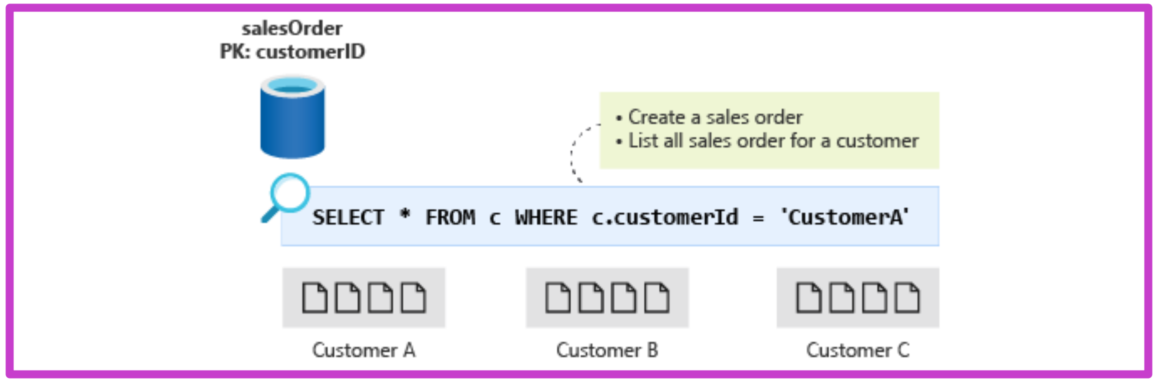
Review your operations and then decide whether to embed or reference your related data.

Model sales order entities



```
{
  "id": "...",
  "customerId": "...",
  "details": [{
    "sku": "...",
    "name": "...",
    "description": "...",
    "price": "...",
    "quantity": "...",
  }]
}
```

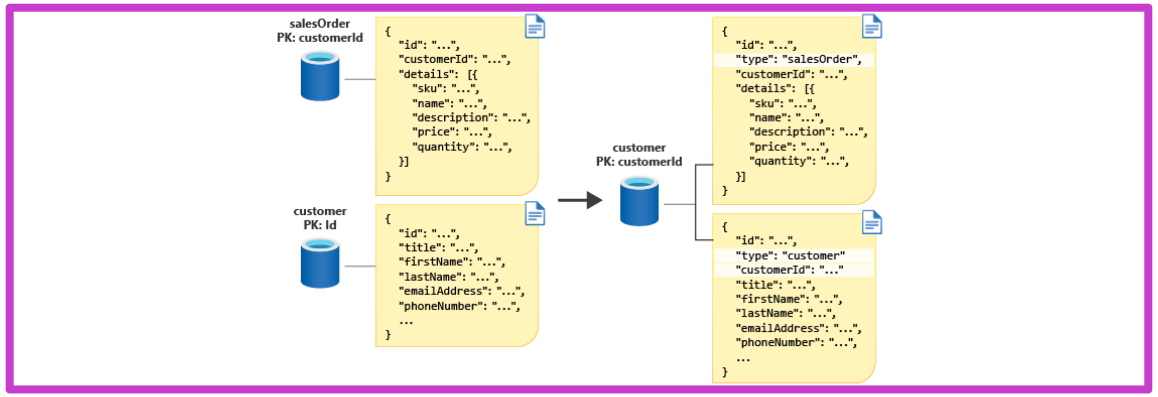
- Create a sales order
- List all sales orders for a customer
- Query top 10 customers by number of sales orders



- Create a sales order
- List all sales order for a customer

```
SELECT * FROM c WHERE c.customerId = 'CustomerA'
```

Identify optimization opportunities

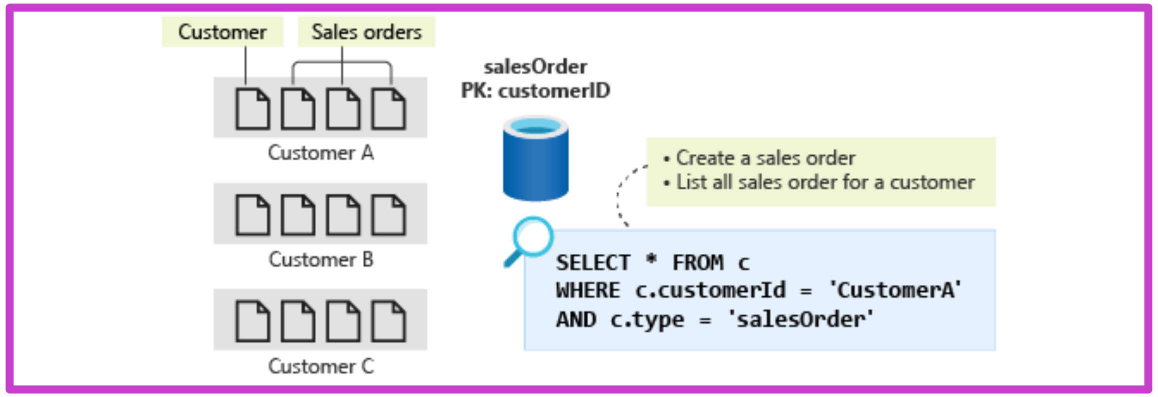


```
{
  "id": "...",
  "customerId": "...",
  "details": [{
    "sku": "...",
    "name": "...",
    "description": "...",
    "price": "...",
    "quantity": "...",
  }]
}
```

```
{
  "id": "...",
  "title": "...",
  "firstName": "...",
  "lastName": "...",
  "emailAddress": "...",
  "phoneNumber": "...",
  ...
}
```

```
{
  "id": "...",
  "type": "salesOrder",
  "customerId": "...",
  "details": [{
    "sku": "...",
    "name": "...",
    "description": "...",
    "price": "...",
    "quantity": "...",
  }]
}
```

```
{
  "id": "...",
  "type": "customer"
  "customerId": "..."
  "title": "...",
  "firstName": "...",
  "lastName": "...",
  "emailAddress": "...",
  "phoneNumber": "...",
  ...
}
```

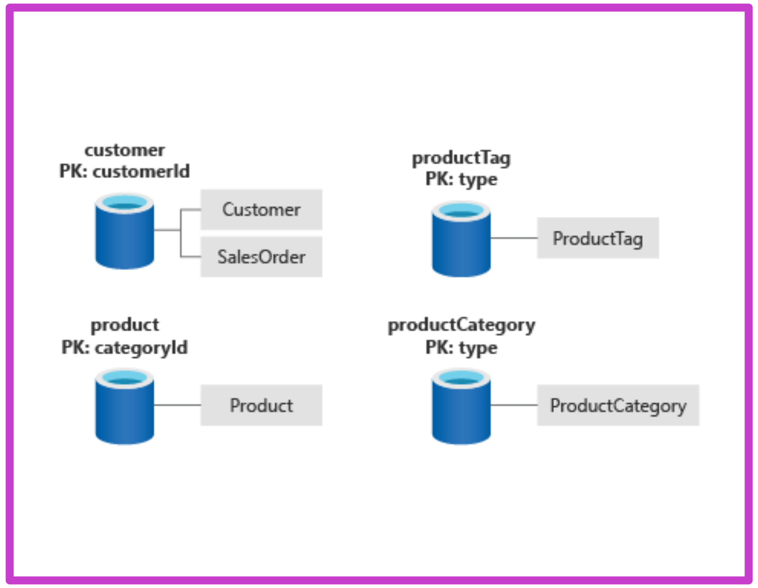


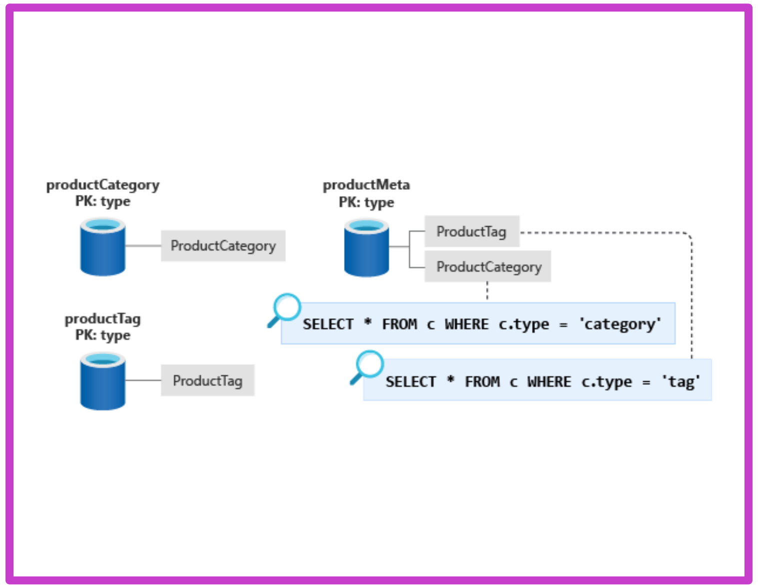
- Create a sales order
- List all sales order for a customer

```
SELECT * FROM c
WHERE c.customerId = 'CustomerA'
AND c.type = 'salesOrder'
```

Finalize the data model

You've nearly finished remodeling your database. You've transformed nine relational database tables into four containers for your NoSQL database.

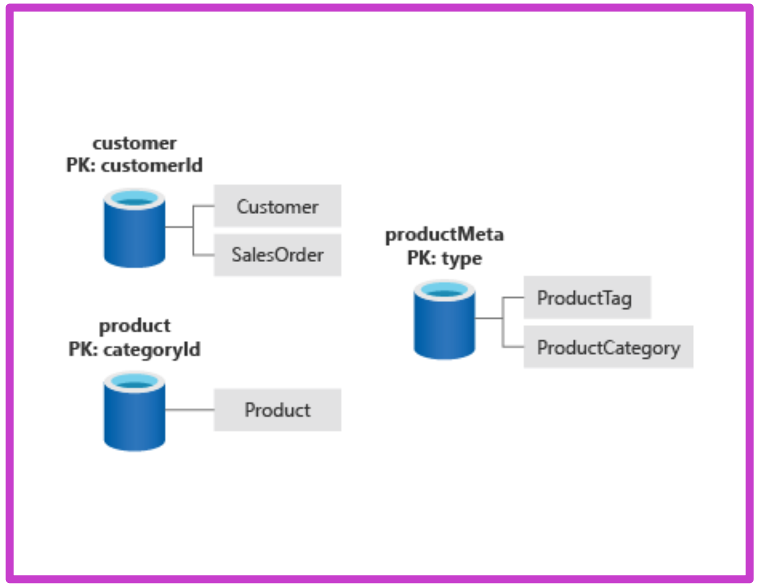




```
SELECT * FROM c WHERE c.type = 'category'
```

```
SELECT * FROM c WHERE c.type = 'tag'
```

One final optimization



Your final design

Lab – Cost of denormalizing data and aggregates and using the change feed for referential integrity

- Prepare your development environment
- Exercise 1: Measure performance cost when denormalizing data
- Exercise 2: Use the change feed to manage referential integrity
- Exercise 3: Denormalizing Aggregates
- Start Azure Cloud Shell and open Visual Studio Code
- Complete the code to update total sales orders
- Complete the code to implement transactional batch
- Query for the customer and their sales orders
- Create a new sales order and update total sales orders in a transaction
- Delete an order by using transactional batch
- View the code that deletes a sales order
- View the code for your top 10 customers query

Microsoft