Modeling Relational Data with Cosmos DB

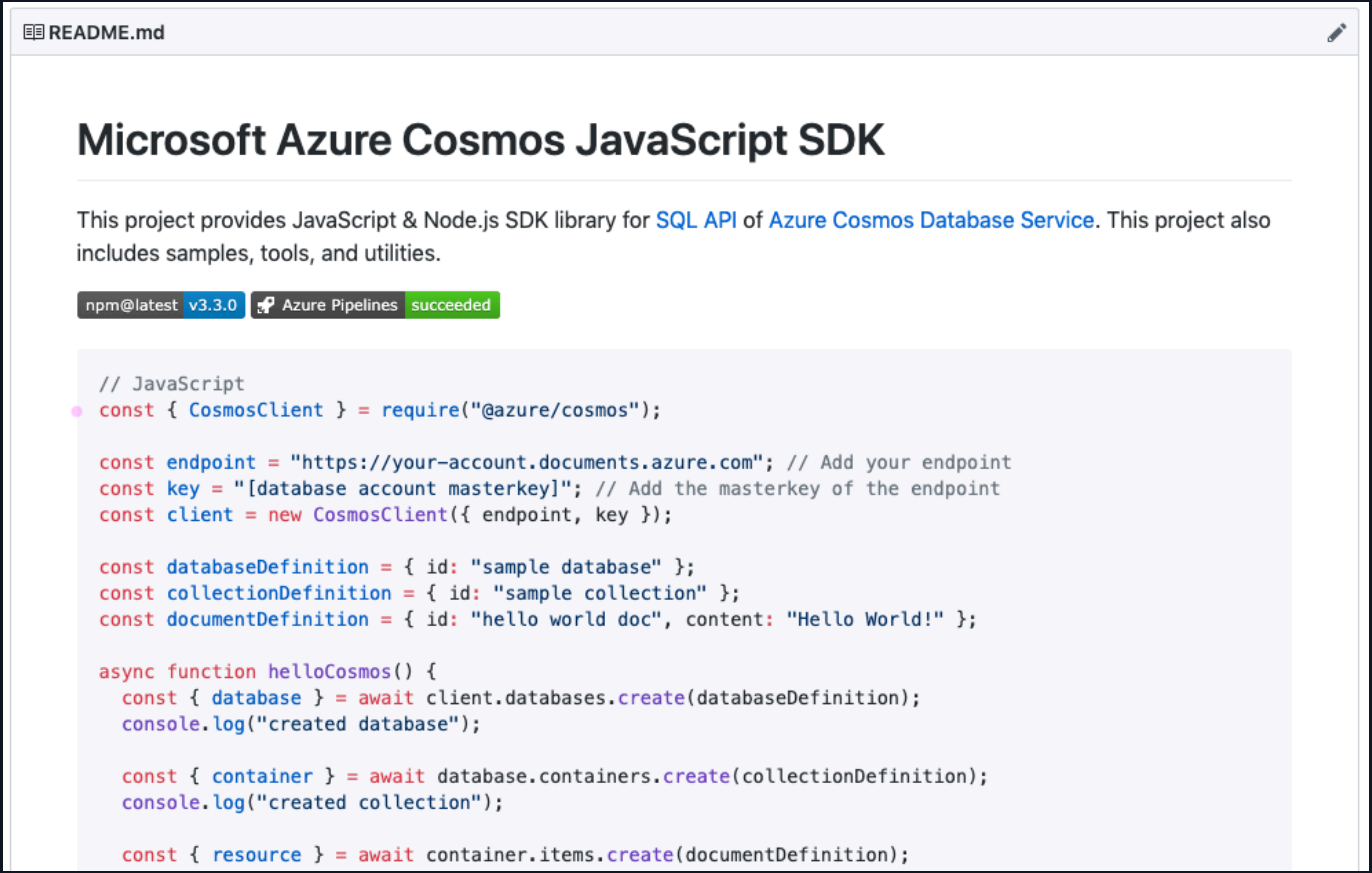
Steve Faulkner

@southpolesteve

AT THE SOUTH POLE!
EPELRIP
SMITH
PMT

Microsoft

Azure Cosmos



README.md

Microsoft Azure Cosmos JavaScript SDK

This project provides JavaScript & Node.js SDK library for SQL API of Azure Cosmos Database Service. This project also includes samples, tools, and utilities.

npm@latest v3.3.0 | Azure Pipelines succeeded

```
// JavaScript
const { CosmosClient } = require("@azure/cosmos");

const endpoint = "https://your-account.documents.azure.com"; // Add your endpoint
const key = "[database account masterkey]"; // Add the masterkey of the endpoint
const client = new CosmosClient({ endpoint, key });

const databaseDefinition = { id: "sample database" };
const collectionDefinition = { id: "sample collection" };
const documentDefinition = { id: "hello world doc", content: "Hello World!" };

async function helloCosmos() {
  const { database } = await client.databases.create(databaseDefinition);
  console.log("created database");

  const { container } = await database.containers.create(collectionDefinition);
  console.log("created collection");

  const { resource } = await container.items.create(documentDefinition);
```

Modeling Relational Data with Cosmos DB

A brief history of databases

1974
SQL Invented

1981
Price per GB?

$700,000

2017

Price per GB?

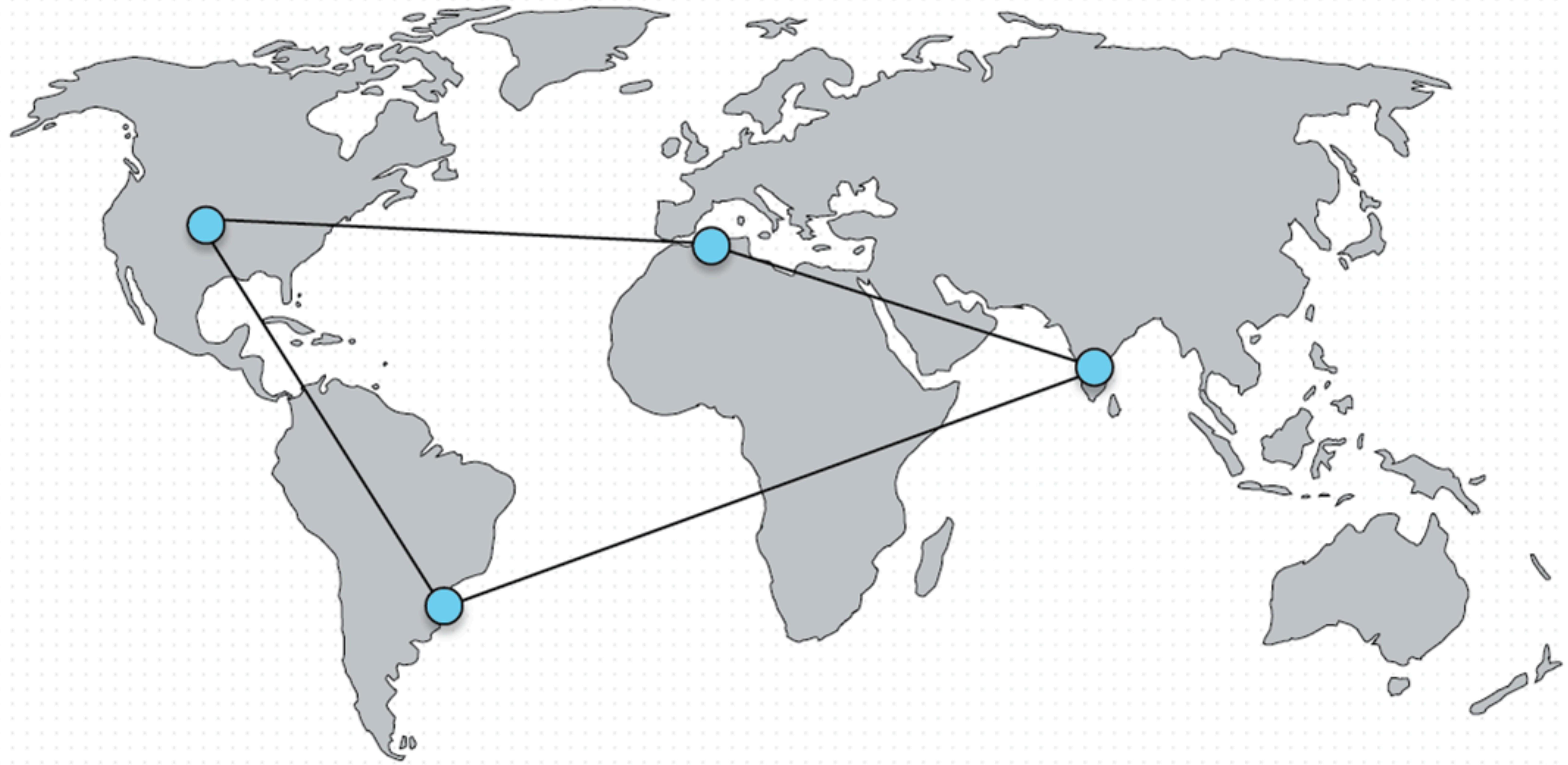
$0.025

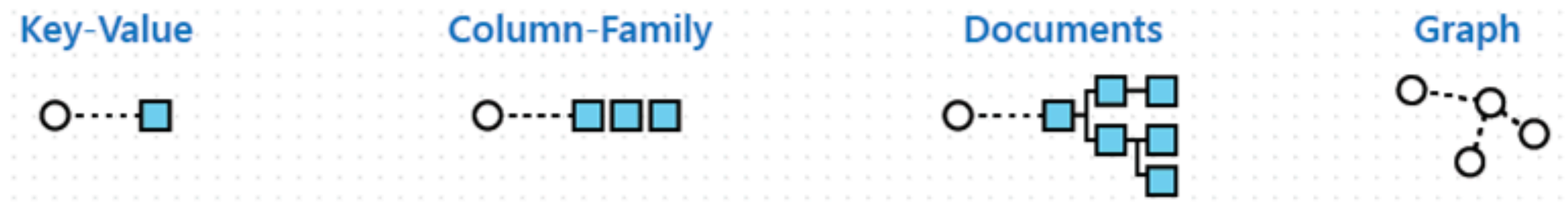
SQL - Storage Optimized
NoSQL - Speed Optimized

Azure Cosmos

Azure Cosmos DB

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





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

Read/Write JSON

Massive scale+perf

No schema

Partitioned / Scale out

SQL API

Everything indexed by default

Multiple APIs

SQL
Tables
Cassandra
Mongo
Gremlin

✨Jupyter Notebooks✨

✨Spark✨

✨etcd✨

Let’s talk about “SQL”

SQL <u>API</u>
Not a SQL Database

Queries

```
SELECT * from c
SELECT c.id from c
SELECT * from c
WHERE c.dueAt > today
```

```
SELECT * from c
WHERE c.priority > 10
ORDER BY c.dueAt
```

SELECT, WHERE, DISTINCT, BETWEEN, IN, TOP, ORDER BY, COUNT, MAX, MIN, AVG

✨Multi ORDER BY✨

✨OFFSET LIMIT✨

✨DISTINCT✨

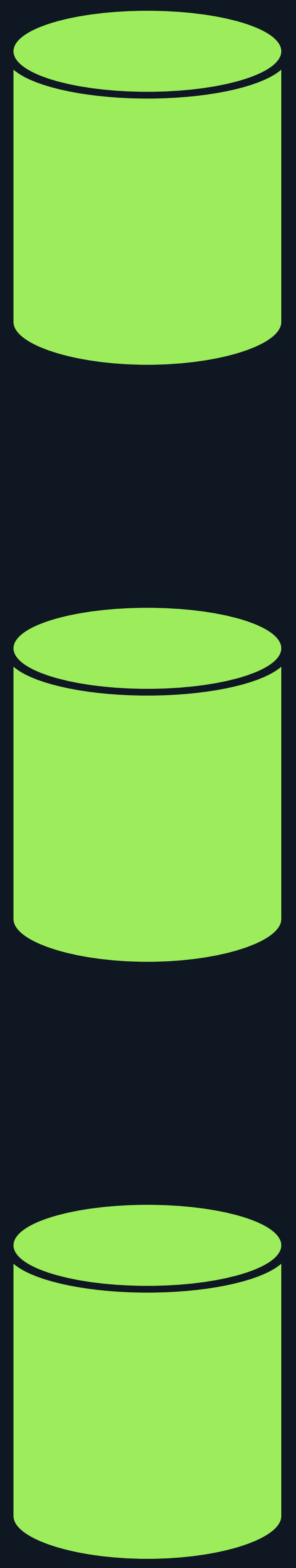
✨Correlated Subqueries✨

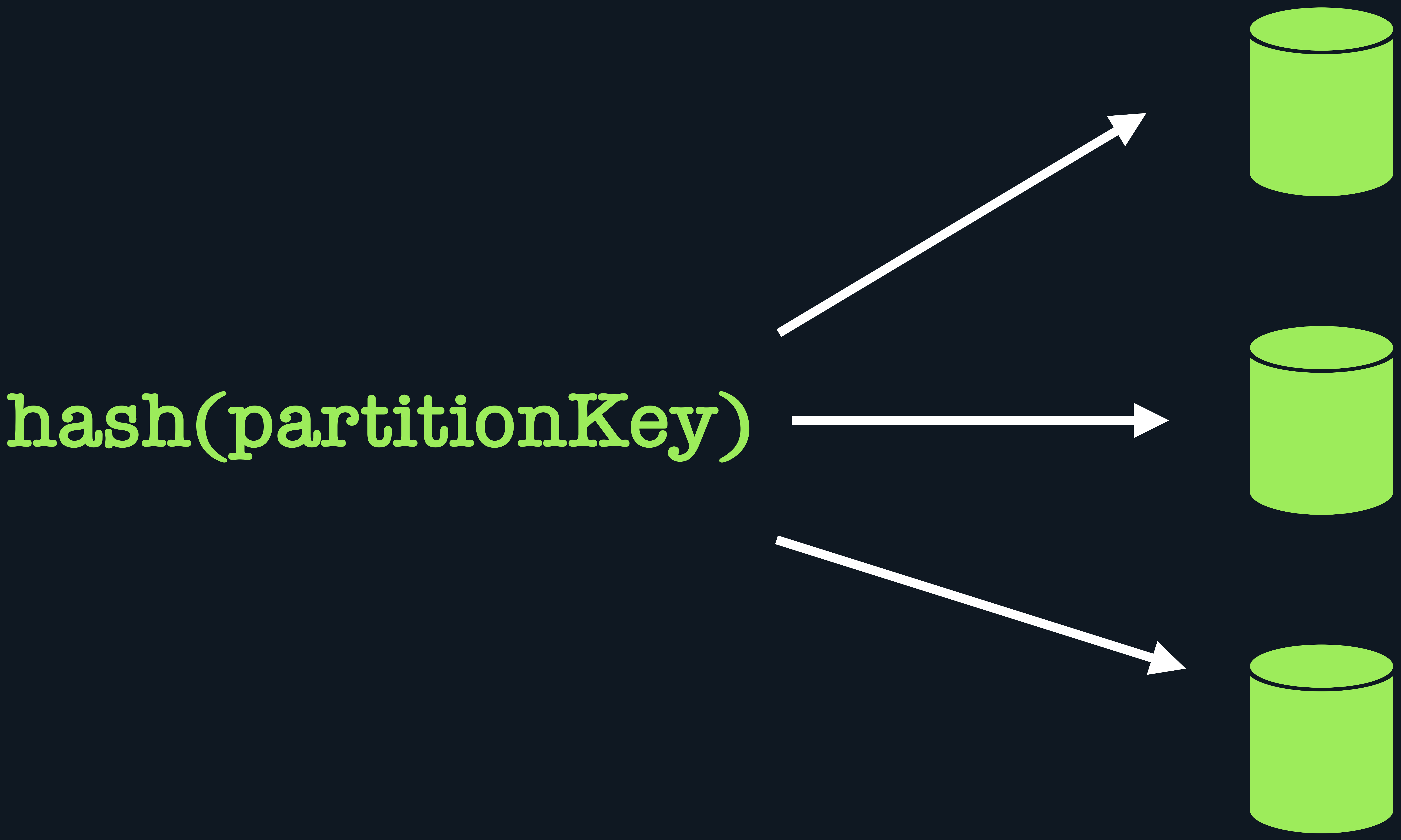
NoSQL Gotchas

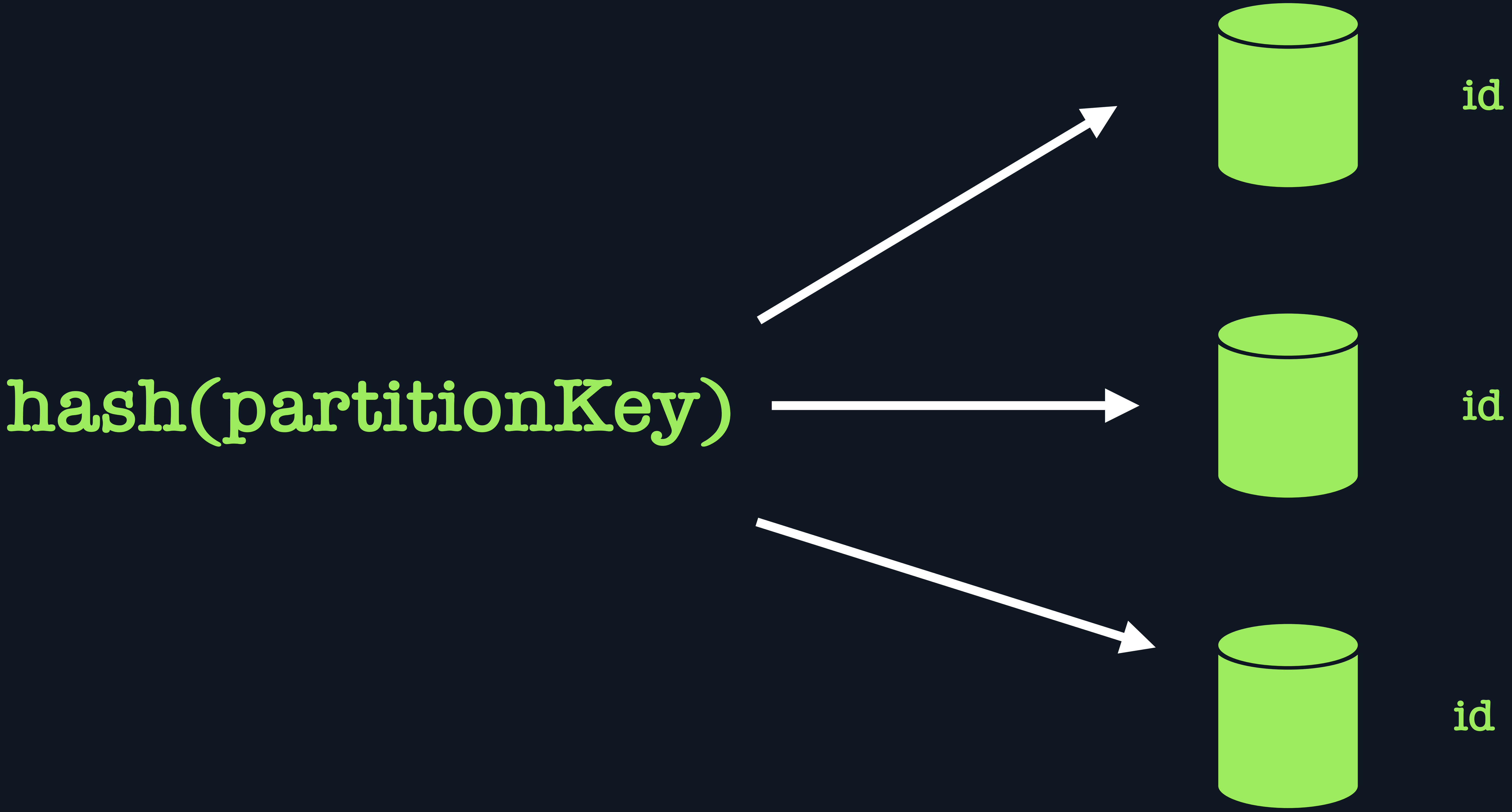
Scans

Cross Partition Queries

Partitioning



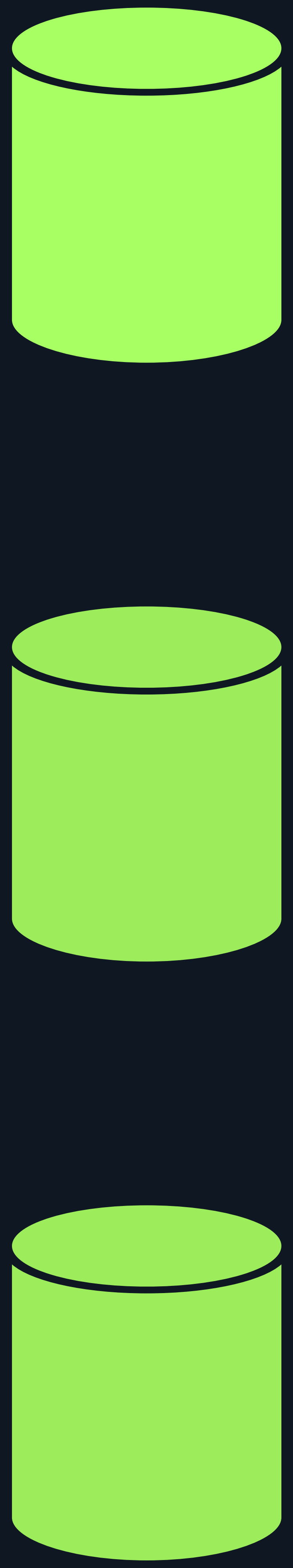

hash(partitionKey)

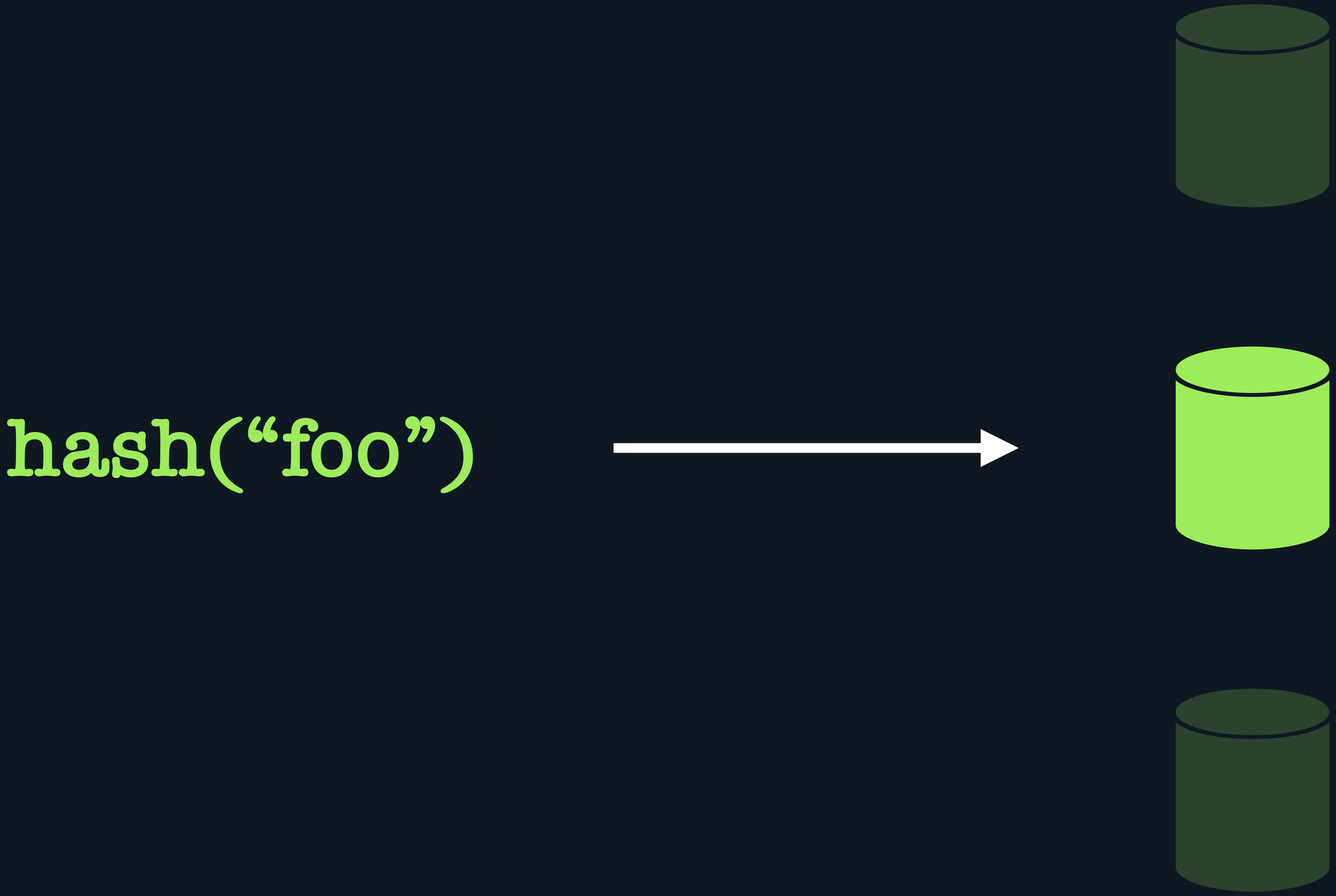

hash(partitionKey)
id
id
id

`id + partitionKey`

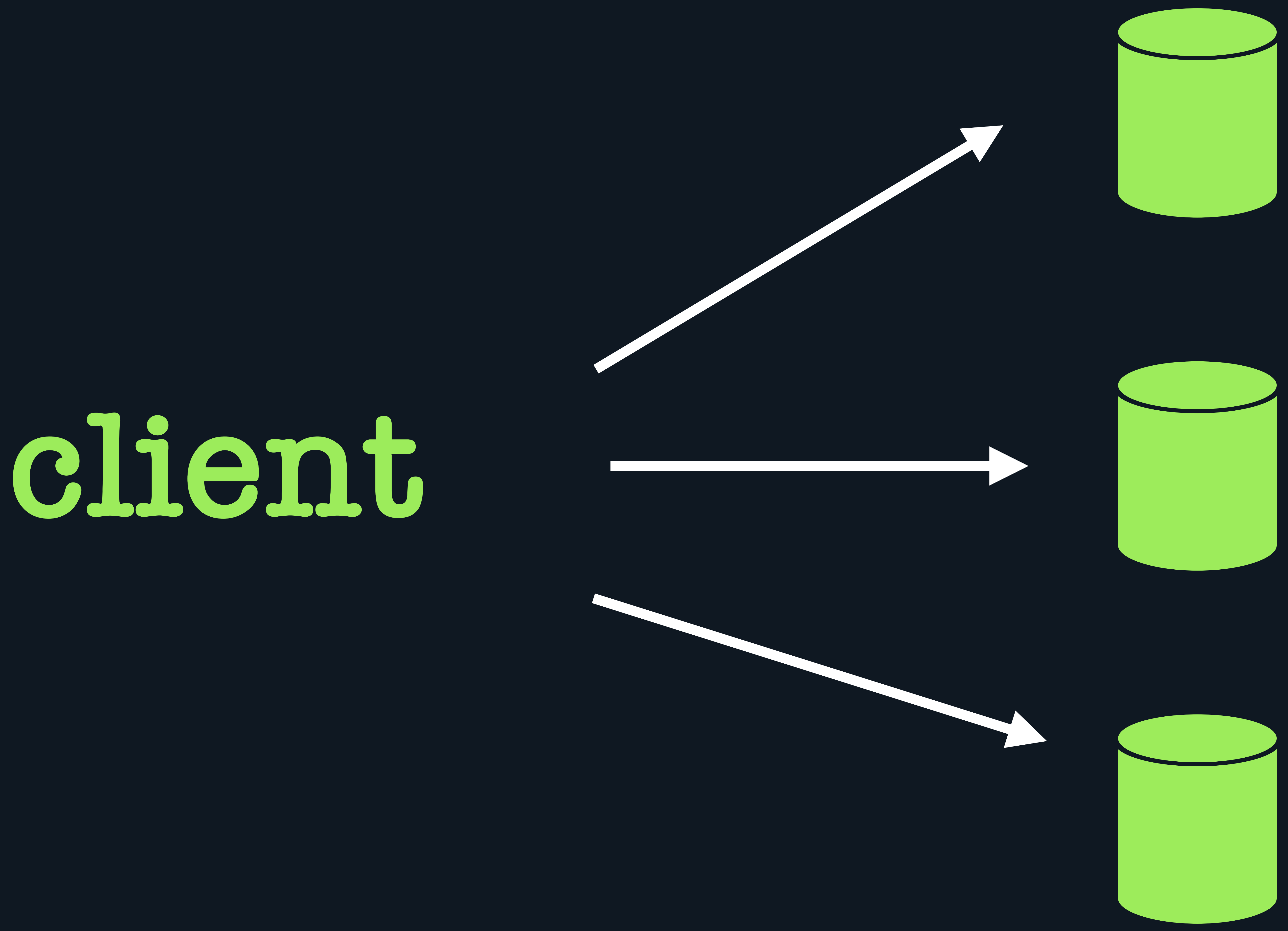
```
SELECT * from c WHERE
c.partitionKey = “foo”
```

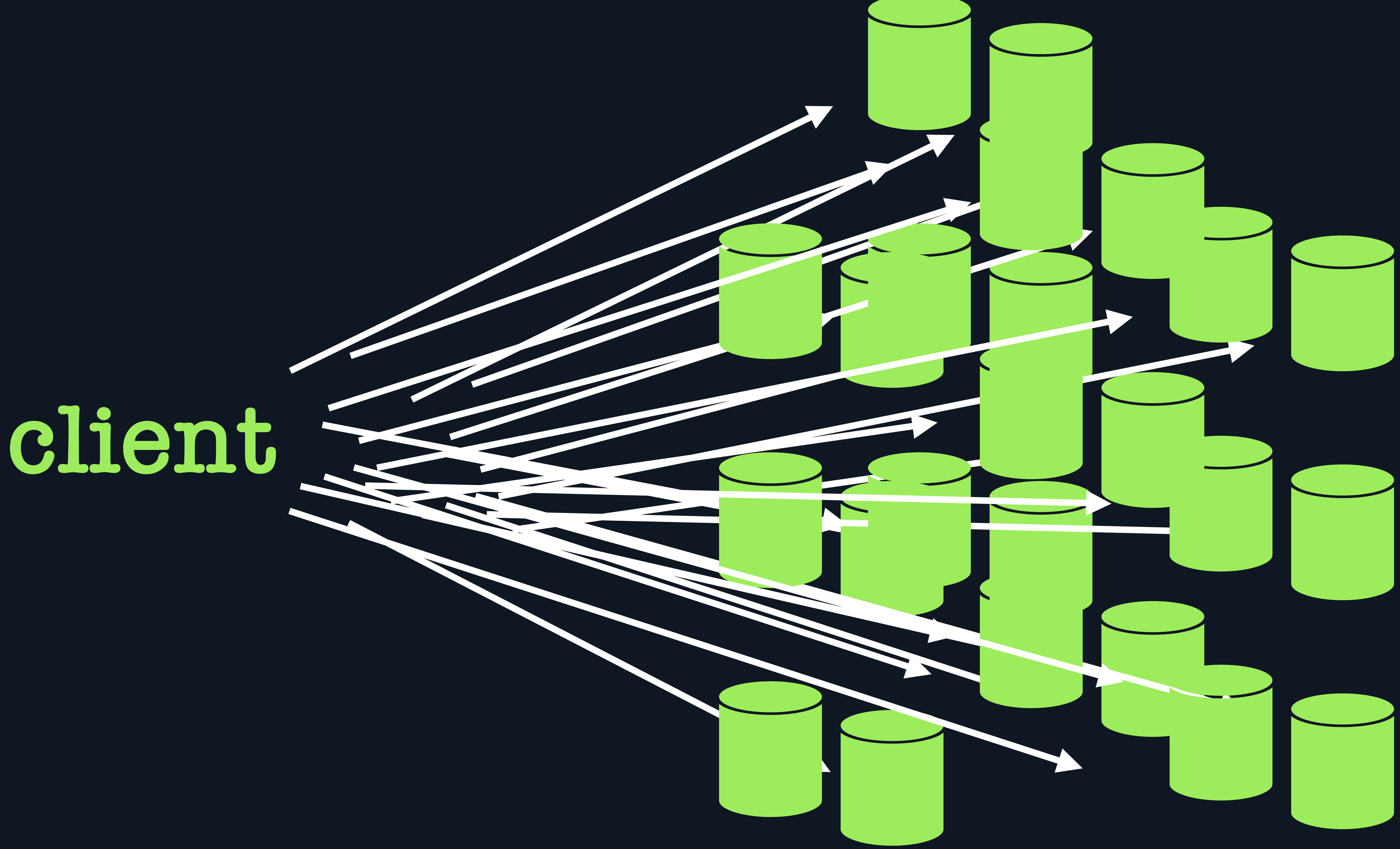
`hash("foo")`

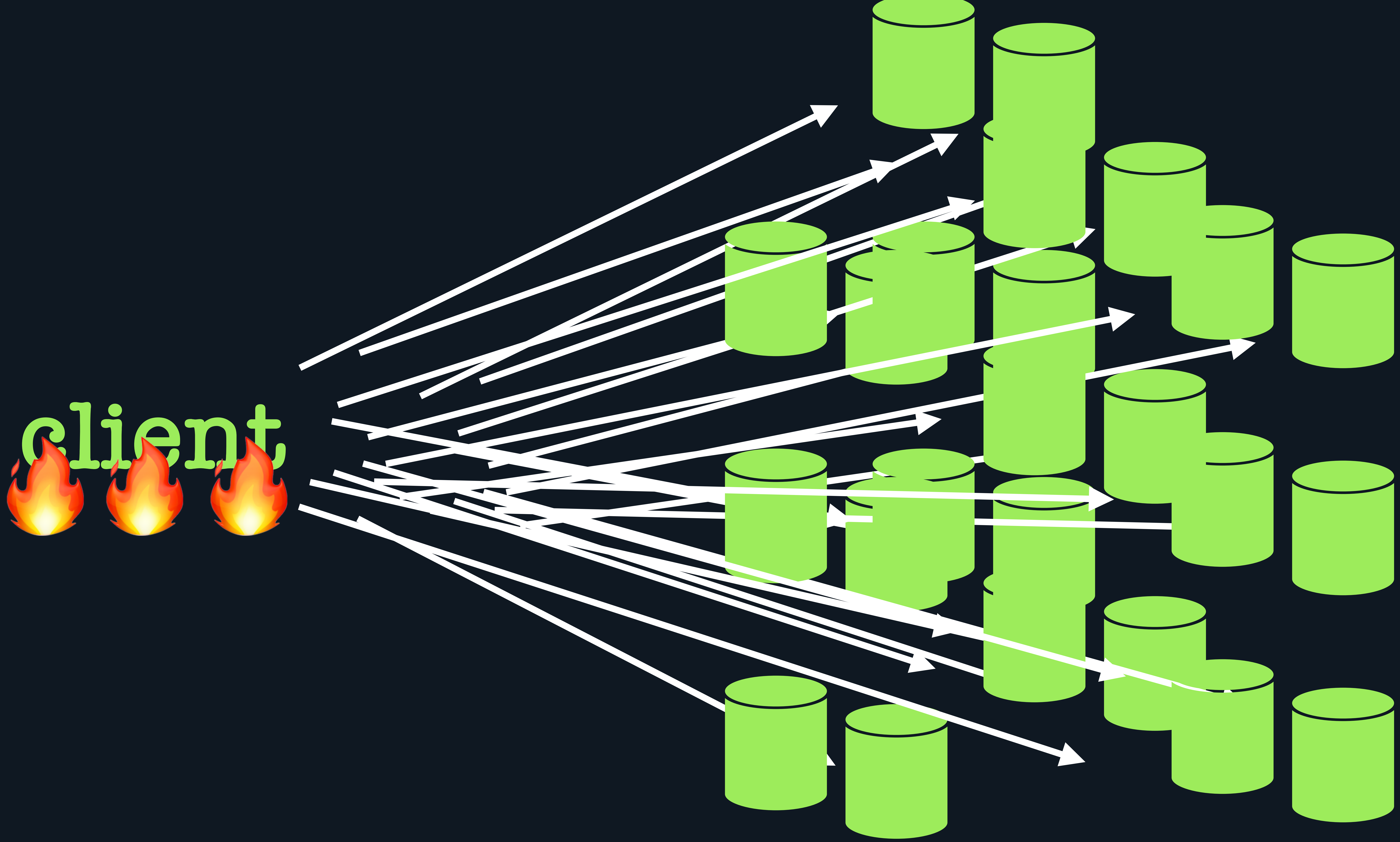



hash("foo")

```
SELECT * from c WHERE
c.notPartitionKey = “bar”
```


client


client


client
🔥🔥🔥

RECAP

SQL API != SQL

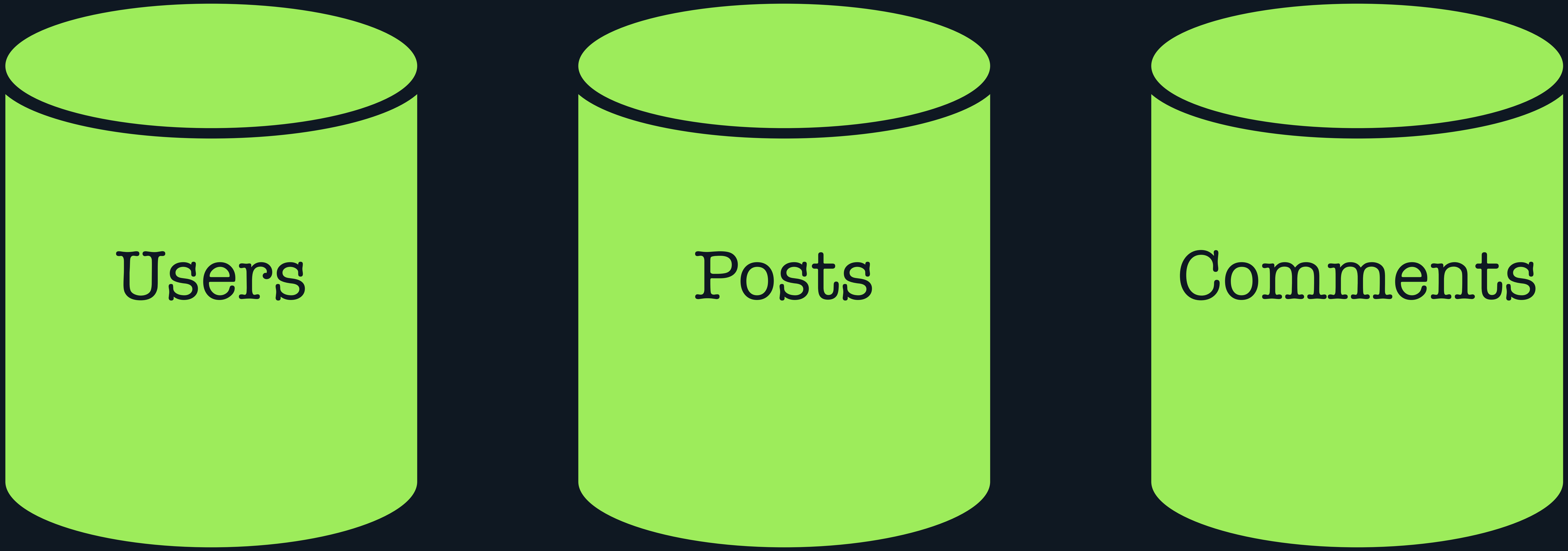
❤️ partition key ❤️

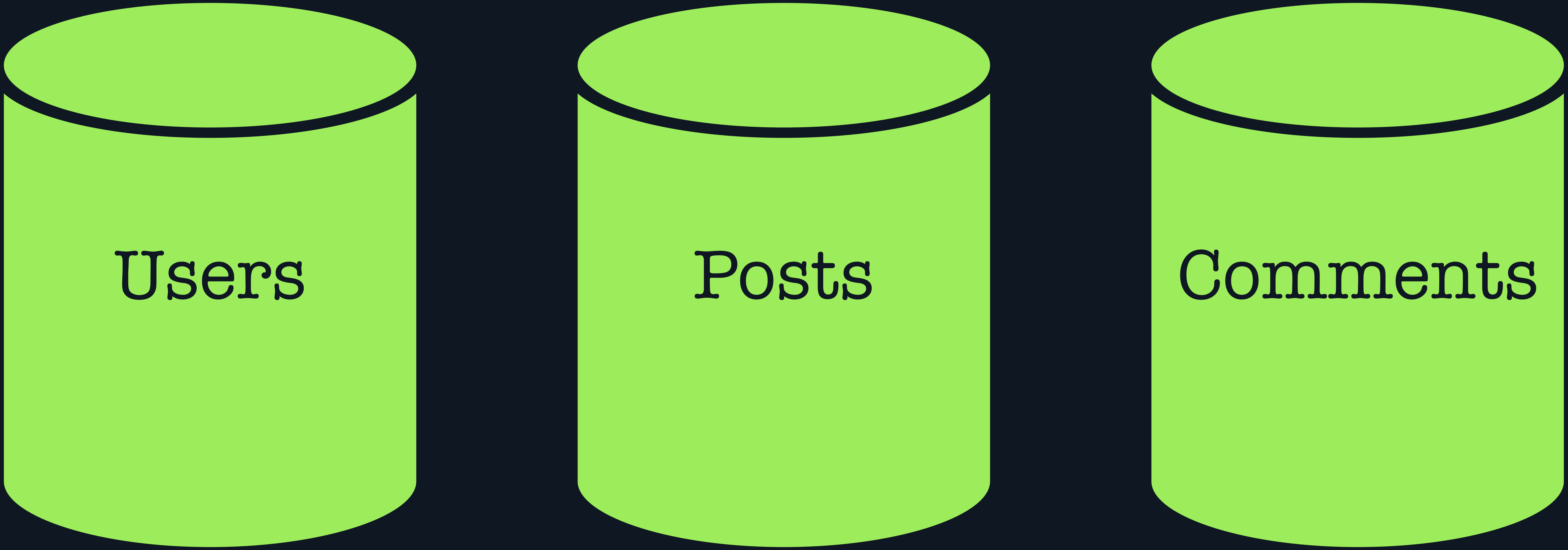
Avoid Cross Partition

Relational Data

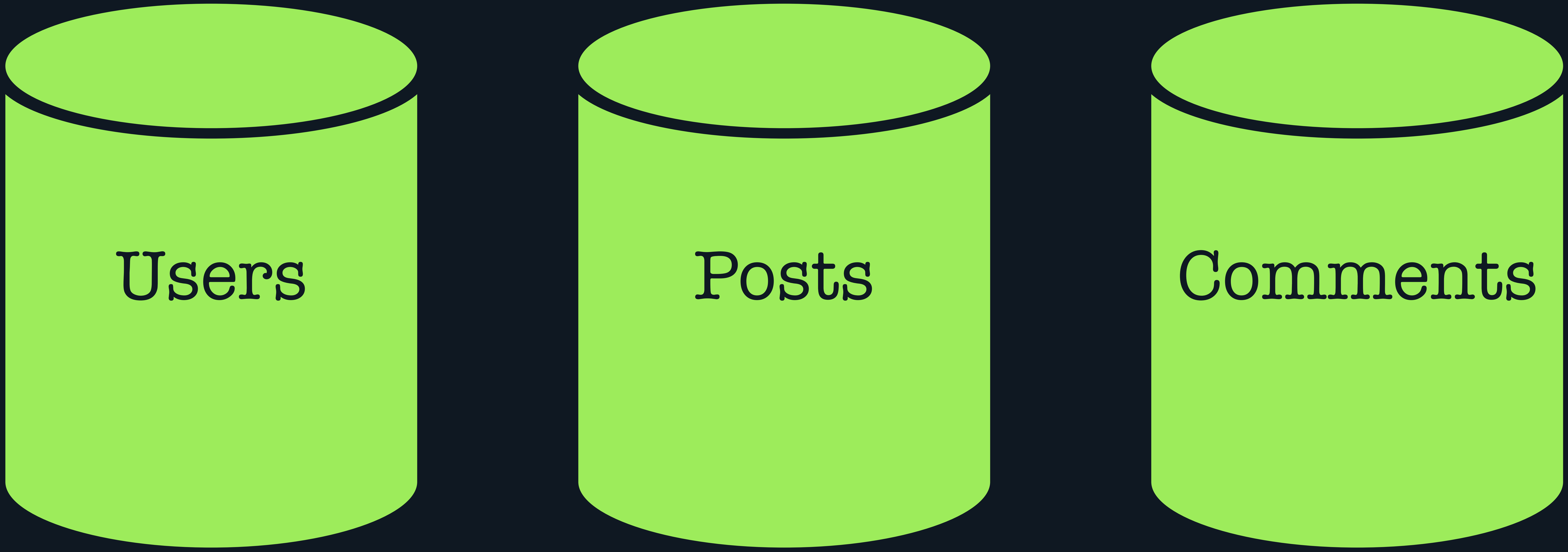
Blog Example

Users
Posts
Comments


Users
Posts
Comments

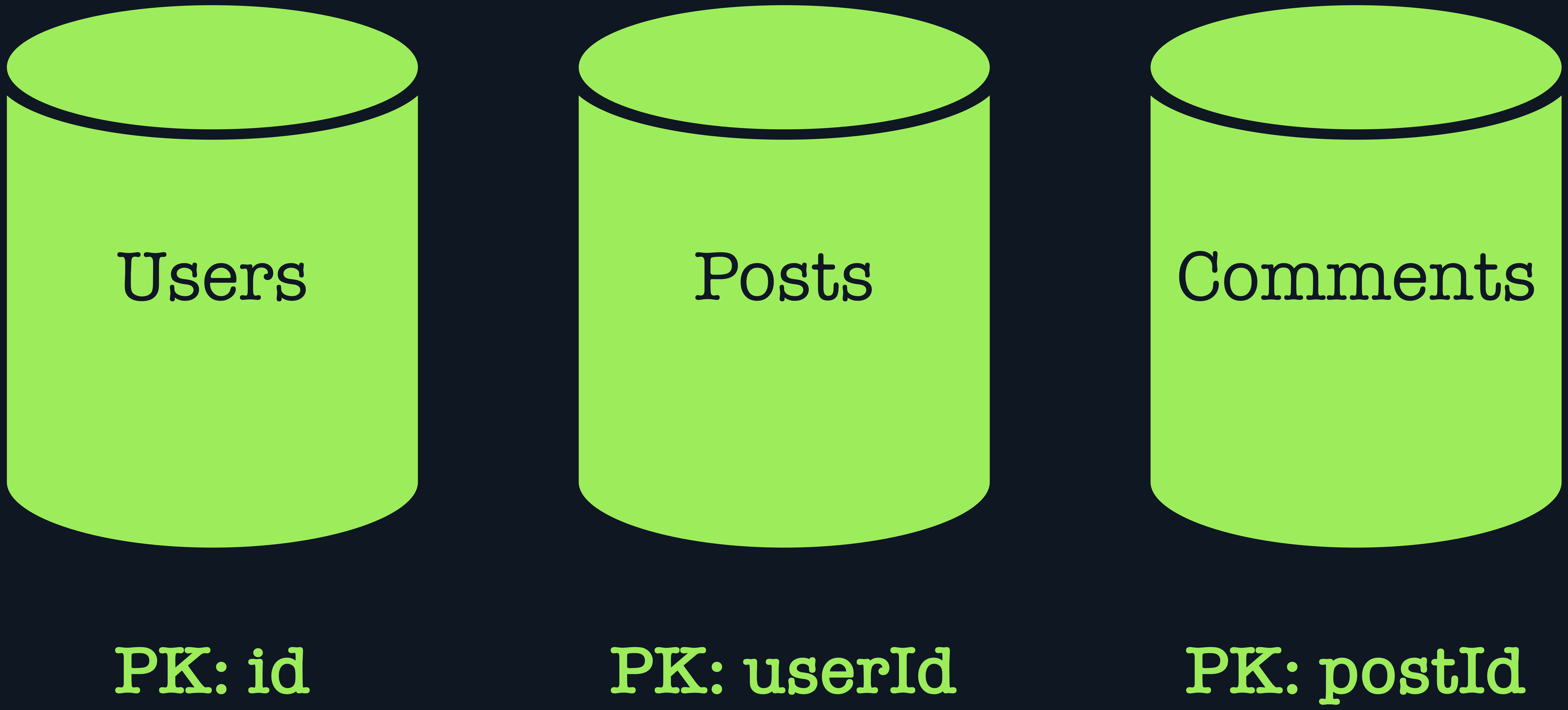


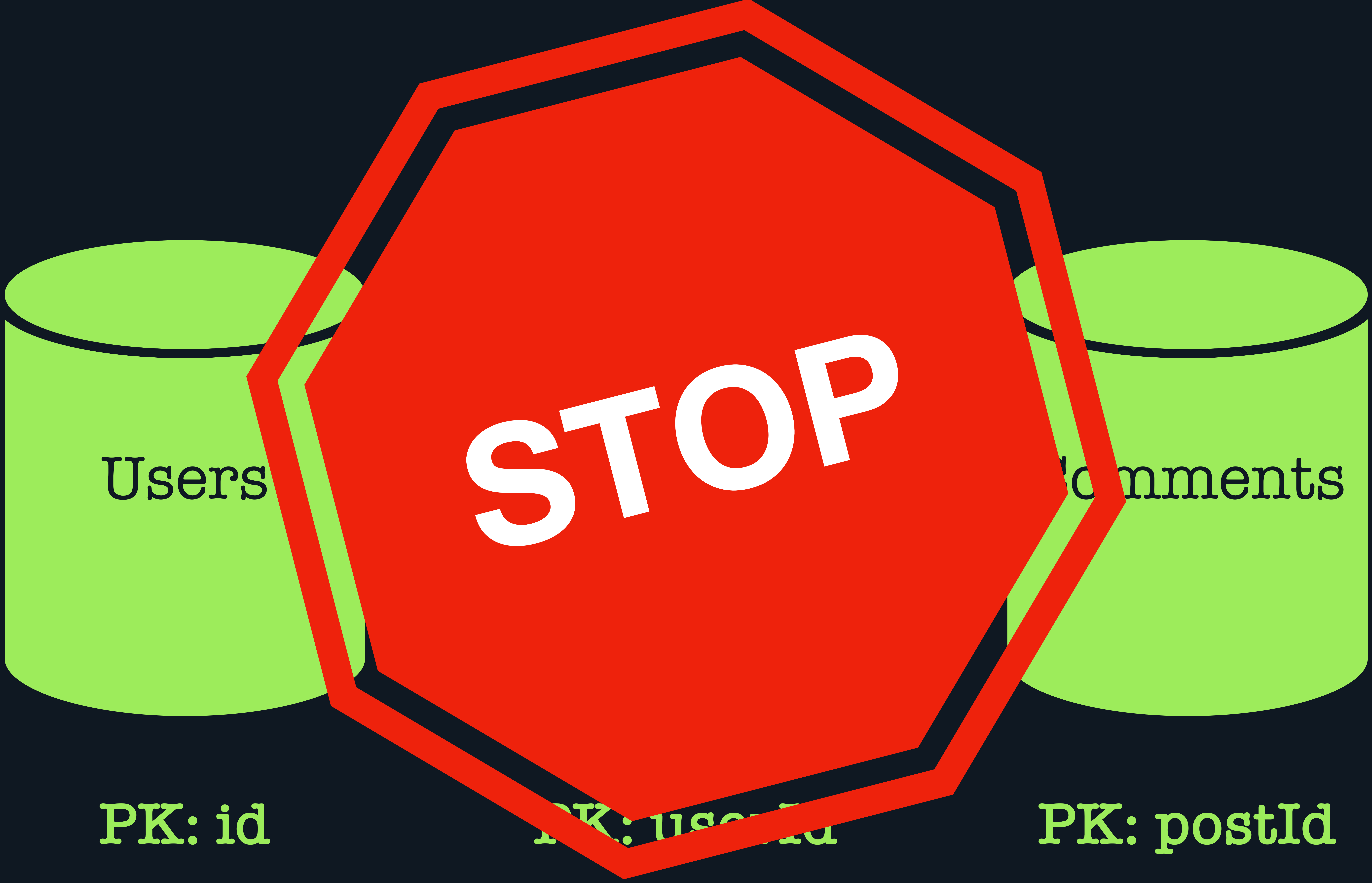
PK: id

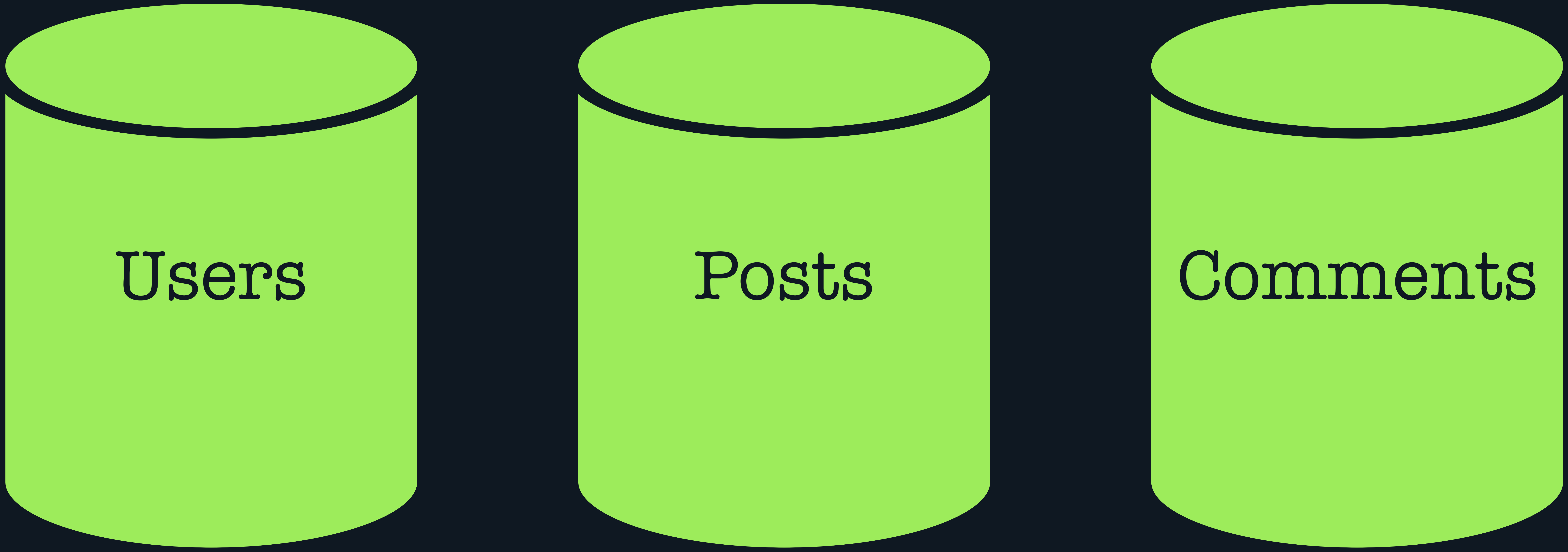


PK: id

PK: userId


Users
Posts
Comments
PK: id
PK: userId
PK: postId


Users
STOP
Comments
PK: id
PK: userId
PK: postId



PK: id

PK: userId

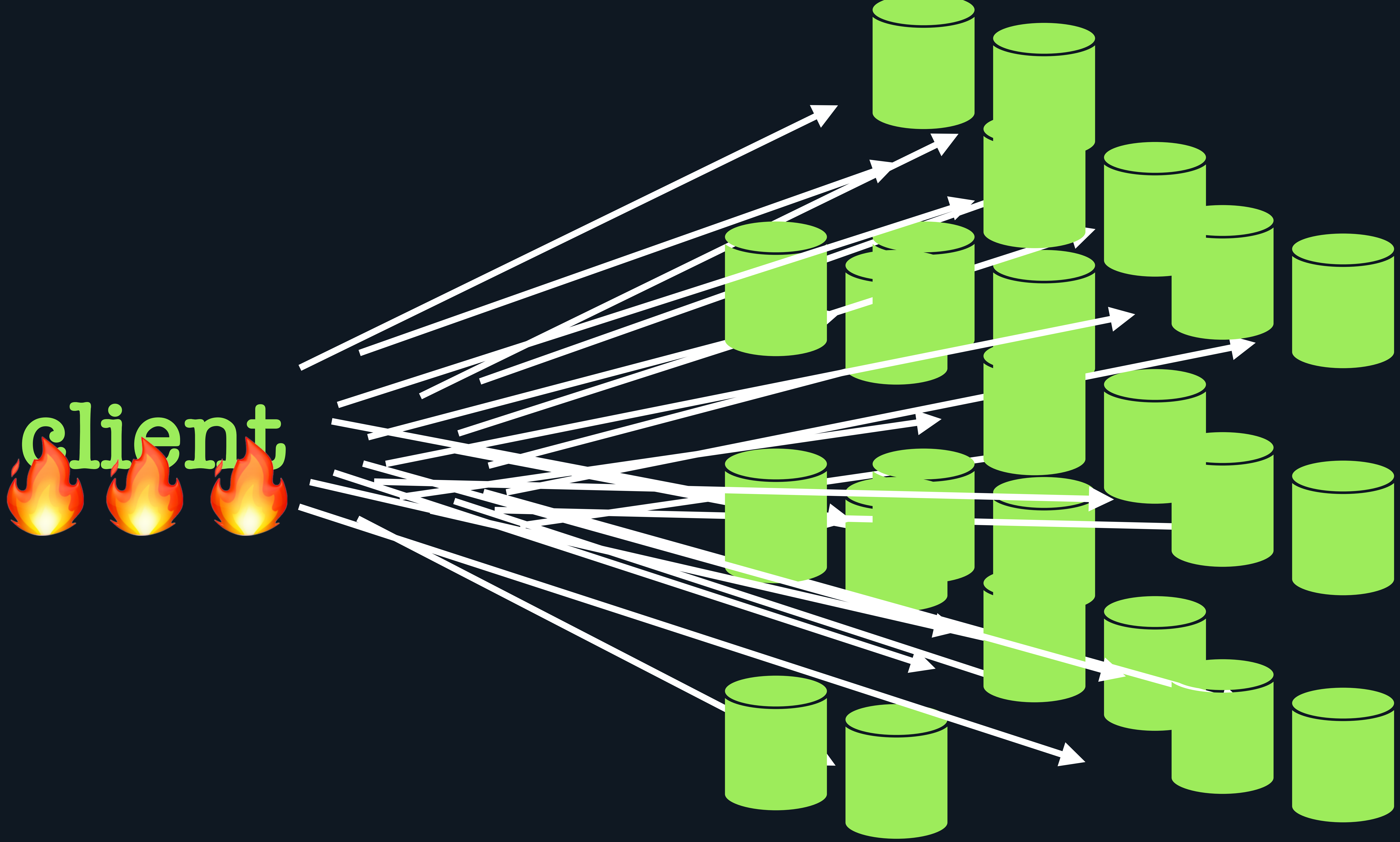
PK: postId

Cross Partition Queries

Posts

pk: userId

```
Post.find(1)
```

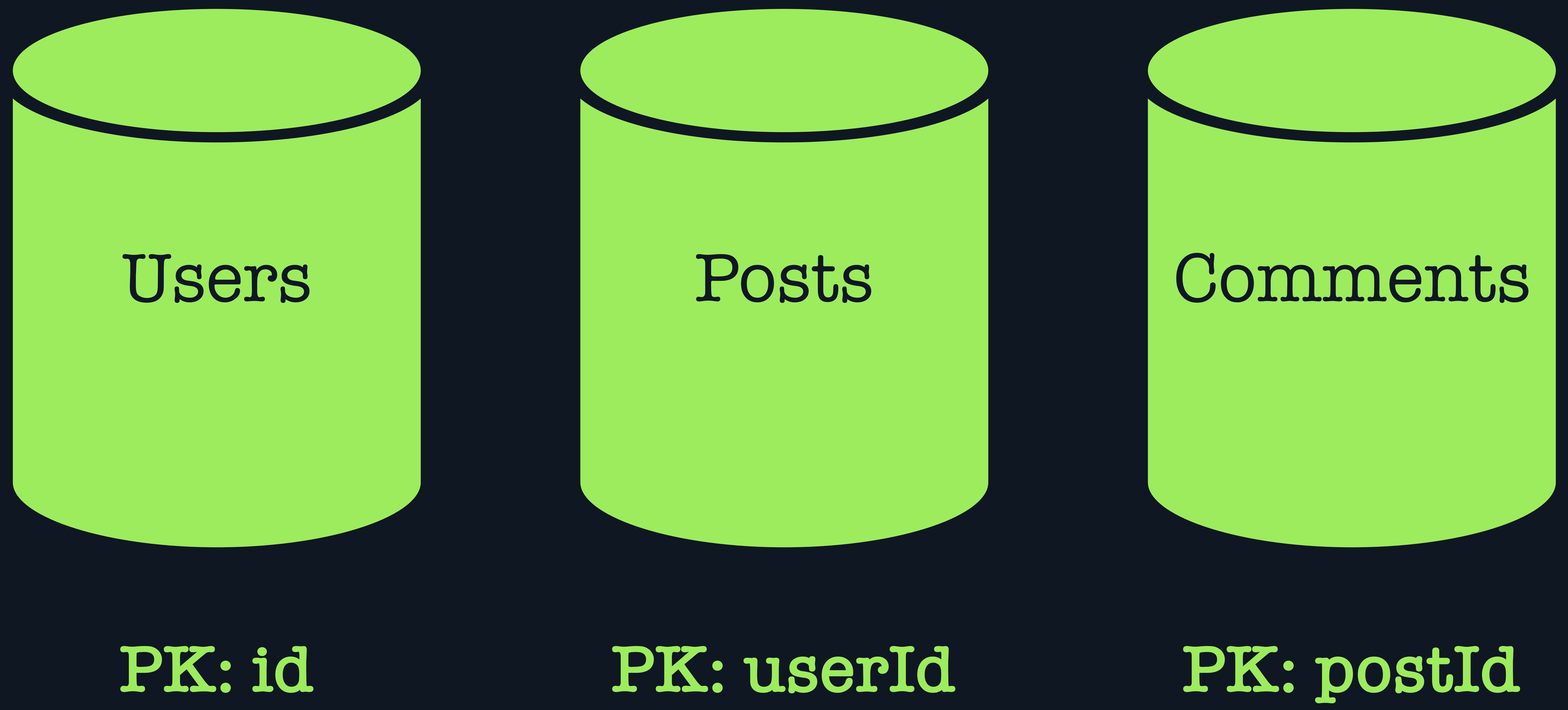

client
🔥🔥🔥

Comments
pk: postId

```
Comment.find(1)
Comments.where(user: 1)
```

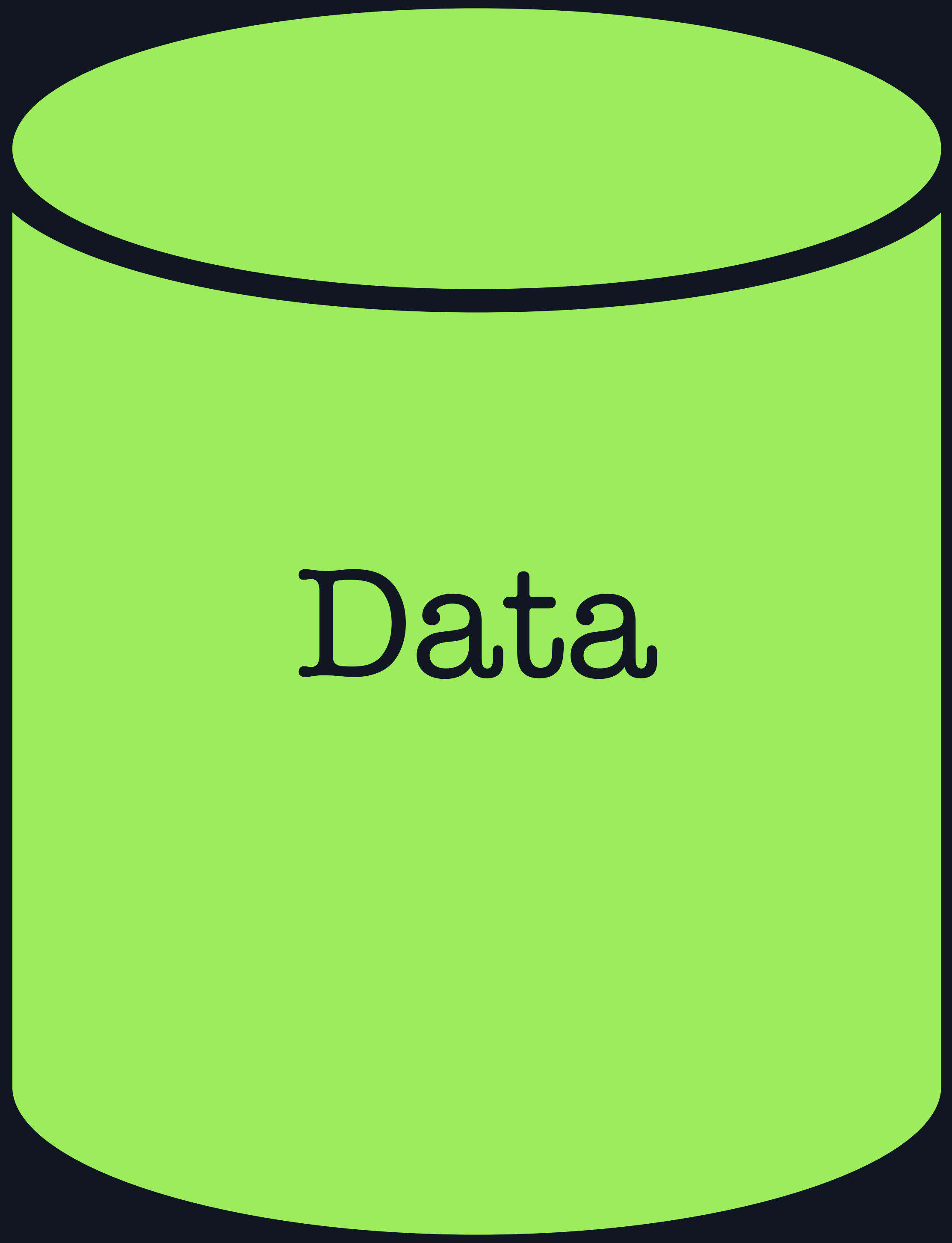
SQL Thinking

Database schema
Normalized data
Rigid consistency
Many tables


Users
Posts
Comments
PK: id
PK: userId
PK: postId

NoSQL Thinking

Application schema
Denormalized data
Flexible consistency
Single container



PK: pk

```
Post.find(1)
```

Post.create()

```
{
  id: “post-1”
  pk: “post-1”
  title: “Cosmos is great!”
}
```

Post.find(1)

```
{
  id: “post-1”
  pk: “post-1”
  title: “Cosmos is great!”
}
```

```
User.find(1).posts()
```

Post.create()

```
{
  id: “post-2”
  pk: “post-2”
  authorId: “1”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
}

{
  id: “post-3”
  pk: “user-1-posts”
}

{
  id: “post-4”
  pk: “user-1-posts”
}
```

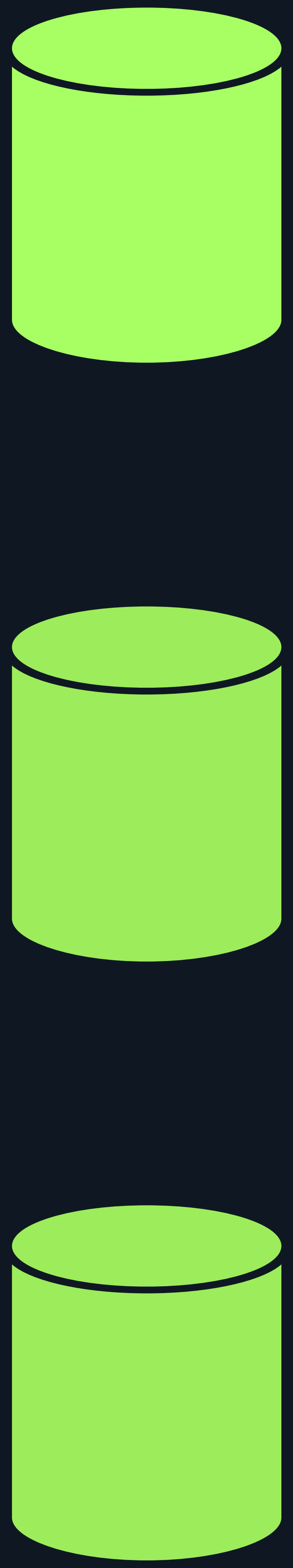
```
User.find(1).posts()
```

```
Select c.id from c WHERE
c.pk = “user-1-posts”
```

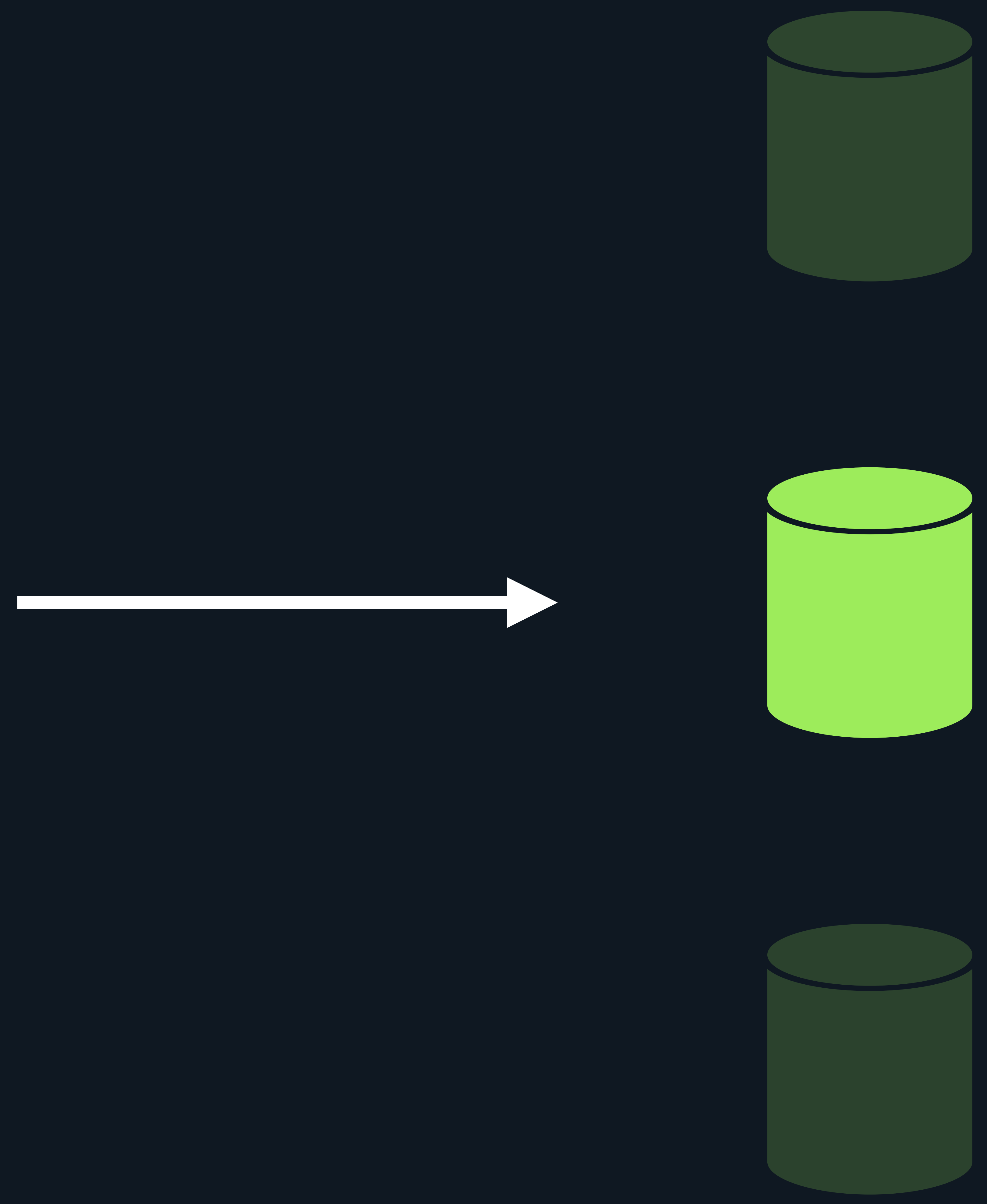
```
User.find(1).posts()
```

```
Select c.id from c WHERE
c.pk = “user-1-posts”
```

hash("user-1-posts")



hash(“user-1-posts”)



User.find(1).posts()

```
[
  { id: “post-2” }
  { id: “post-3” }
  { id: “post-4” }

  …
]
```

```
User.find(1)
.posts({ state: draft })
```

Posts.create()

```
{
  id: “post-2”
  pk: “post-2”
  authorId: “user-1”
  state: “draft”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
  state: “draft”
}
```

Posts.create()

```
{
  id: “post-2”
  pk: “post-2”
  authorId: “user-1”
  state: “draft”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
  state: “draft”
}
```

```
Select c.id from c
WHERE c.pk = “user-1-posts”
AND c.state = “draft”
```

Posts.update()

```
{
  id: “post-2”
  pk: “post-2”
  authorId: “user-1”
  state: “draft”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
  state: “draft”
}
```

Posts.update()

```
{
  id: “post-2”
  pk: “post-2”
  authorId: “user-1”
  state: “published”
}
```

```
{
  id: “post-2”
  pk: “user-1-posts”
  state: “published”
}
```

```
User.findByEmail()
```

User.create

```
{
  id: “2”
  pk: “2”
  email: “steve@microsoft.com”
}
```

```
{
  id: “2”
  pk: “user-email-steve@microsoft.com”
}
```

User.findByEmail()

```
Select c.id from c WHERE
c.email = “steve@microsoft.com”
```

User.findByEmail()

Select c.id from c WHERE
c.pk = “user-emails-steve@microsoft.com”

Advanced Techniques

Update w/ Change Feed + Azure Functions

Materialized Documents

Split Containers

Critical Takeaways

Single Container

Best partition key?
“partitionKey”

Top Level Entities:

id == pk

Use partition key to index relationships and attributes

Optimize as data size grows and access patterns evolve

Questions?

@southpolesteve
askcosmosdb@microsoft.com