dev up

Cosmos DB Performance Tuning

Mike Benkovich
Microsoft Azure MVP
mike@benko.com | www.benkoTIPS.com | @mbenko

SyllogisTeks

ArchitectNow

3Cloud

slalom

((CENTRIC))

CONCERO

Microsoft Azure

Edward Jones

REPLY

MiTek

OAKWOOD

VantageLinks

ADAPTIVE SOLUTIONS GROUP

Covenant TECHNOLOGY PARTNERS

TRIZETTO Provider Solutions
A Cognizant Company

SecureIdeas
professionally evil

SBS CREATIX
IT CONSULTANTS

NISA

aspire
A ServiceTitan Company

Progress

Talentporte

HUNTER Engineering Company

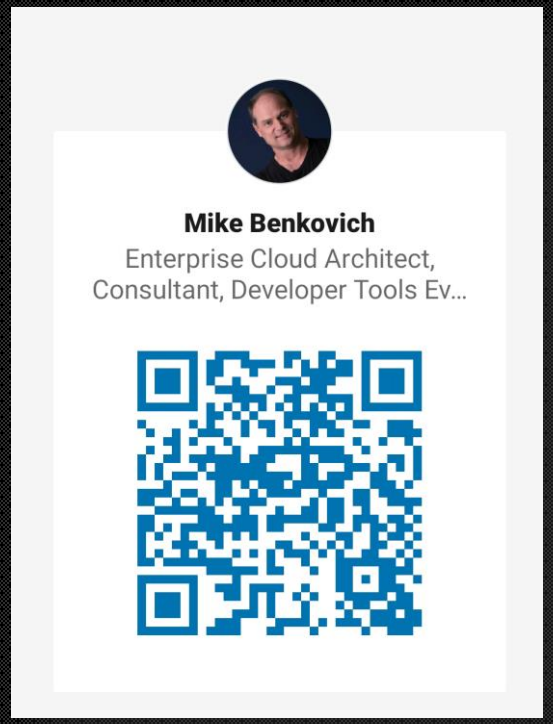
TEALS Program
Building equitable, inclusive computer science programs in high schools
Microsoft

ungerboeck

ais APPLIED INFORMATION SCIENCES

Mike Benkovich

- **Developer**
- Cloud **Architect** & **Consultant**
- Live in **Minneapolis**
- Founder of **Imagine Technologies**, Inc.
- Developing Courses for **LinkedIn** Learning
- Blog www.**benkoTIPS**.com
- Follow **@mbenko** on **Twitter**
- Send me **Feedback**! mike@benkotips.com



My Sessions … this week!

- Permit to Cloud – Land with Confidence in Azure
  - Tuesday 6/7 11:00 am – Discovery D
- Performance Tuning Strategies for Cosmos DB
  - Tuesday 6/7 4:00 pm – Imagination B
- Infrastructure as Code Bake-off ARM vs Bicep vs TF
  - Wednesday 6/8 4:00 pm – Discovery B

What we'll cover today

Why Cosmos
Provisioning Cosmos
Working with Data
Data Modeling Strategies
Performance and tuning

Cosmos DB Overview

A Simple **Global** Assigned NoSQL Database that can be replicated across regions with a click of a button

High **Scalability** and **Availability**

Guaranteed **low latency** and **provisioned** performance

Multi-**model** Multi-**API**

Configurable **Consistency** Options

Containers and Documents

- Database as collection of Documents in Containers
- Provisioned performance – RU's
- Stored as JSON Document
  - Document Explorer (portal)
  - Storage Explorer
- Supports multiple Record Types in Containers
- No referential rules or constraints
- No enforcement of data types


```
{
  "id": "a40d0078",
  "Title": "Add stuff to the Queue",
  "Description": null,
  “Type": “Chore",
  "Owner": null,
  "IsDone": false,
  "CreatedAt": "2019-09-05T11:40:12.4466765Z",
  "CompleteAt": null,
  "_rid": "vr0UAPQ9W4sDAAAAAAAAAA==",
  "_self": "dbs/vr0UAA==/colls/...",
  "_etag": "\"38001cae-0000-0300-0000-5d70...",
  "_attachments": "attachments/",
  "_ts": 1567683617
}
```

Some Terminology

RELATIONAL

- Database
- Table(s), View(s)
- Row
- Column
- Foreign Key
- Join
- Sharding Key

COSMOS DB (SQL API)

- Account/Database
- Container/Collection
- Item/Document (JSON)
- Property
- Reference
- Embedded Document
- Partition Key

What we'll cover today

Why Cosmos

Provisioning Cosmos

Code patterns & tools

Data Modeling Strategies

Performance and tuning

Provisioning Cosmos

Selecting the API : SQL, Mongo, Cassandra, Gremlin, Table

Capacity model : Provisioned vs Serverless

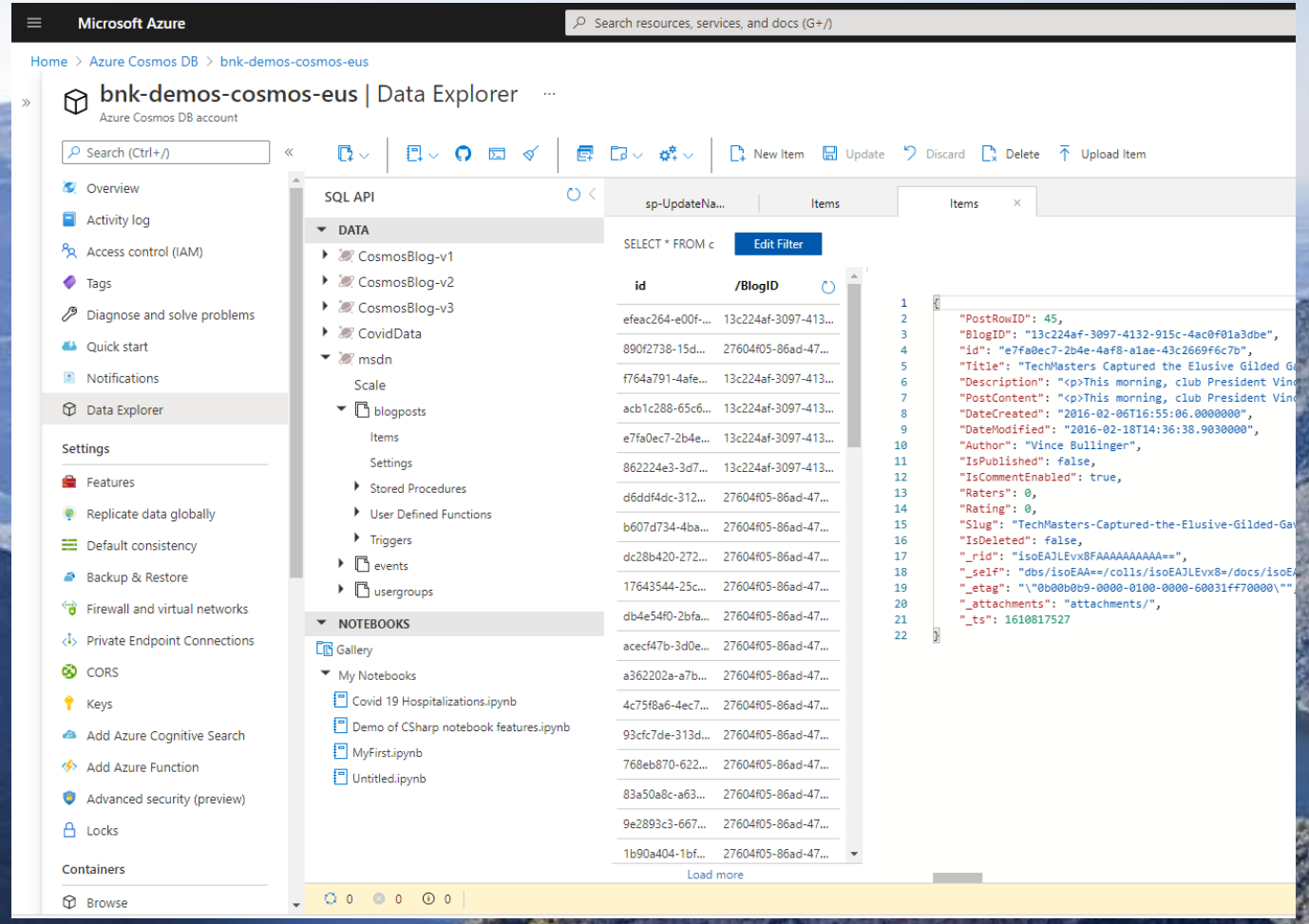
Geo-Redundancy and Multi-region Writes

Networking

Backup

Encryption

Get Started with Cosmos



What we'll cover today

Why Cosmos

Provisioning Cosmos

Working with Data

Data Modeling Strategies

Performance and tuning

Tools to work with Cosmos Data

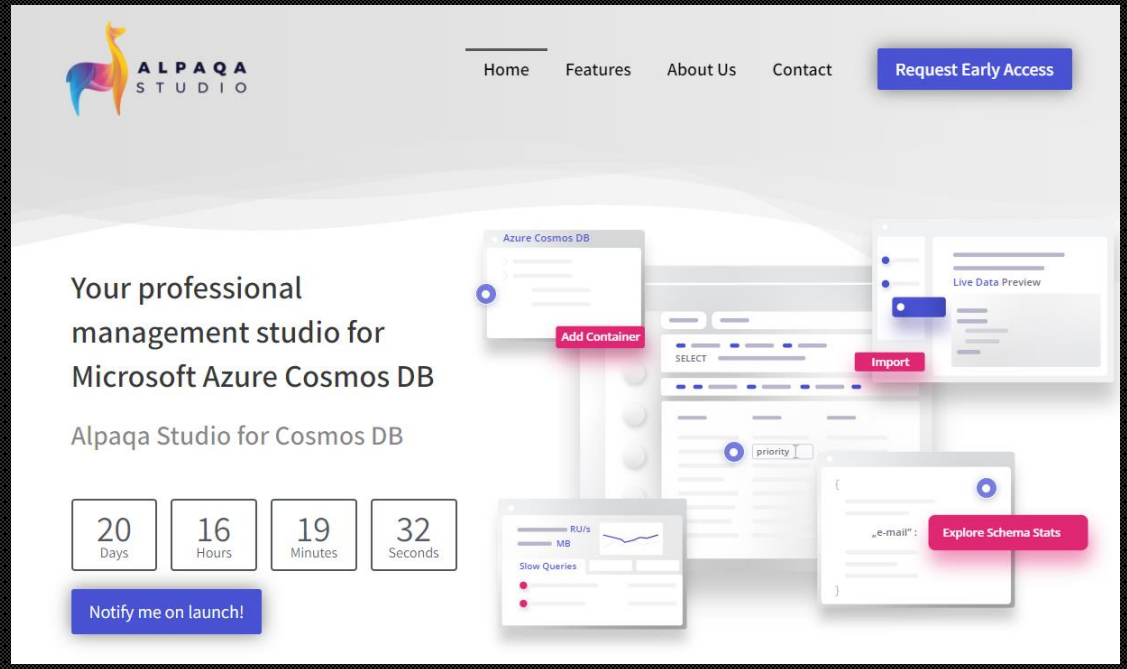
Azure Portal

- Data Explorer
- Jupyter Notebooks

Alpaqa Studio

- Editing Data (in place and bulk)
- Explore Schema
- Background task engine

SDK – Code



https://alpaqastudio.com

[Alpaqa Studio - The IDE for Cosmos DB](#) (preview)

What we'll cover today

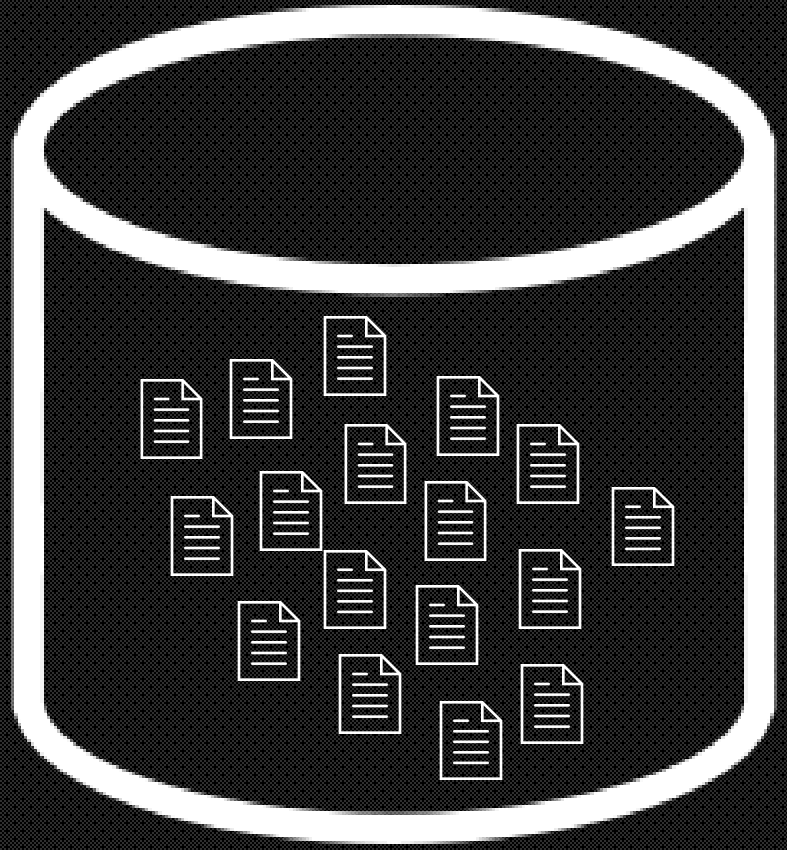
Why Cosmos
Provisioning Cosmos
Working with Data
Data Modeling Strategies
Performance and tuning

Provisioned Performance

- Expected performance has to be **provisioned**
- Expressed in **Request Units per second** (RU/s)
  - Represents the "cost" of a request in terms of CPU, memory and I/O
- Performance can be provisioned:
  - at the database-level
  - at the collection-level
- Provisioned performance **can be changed programmatically** with API calls

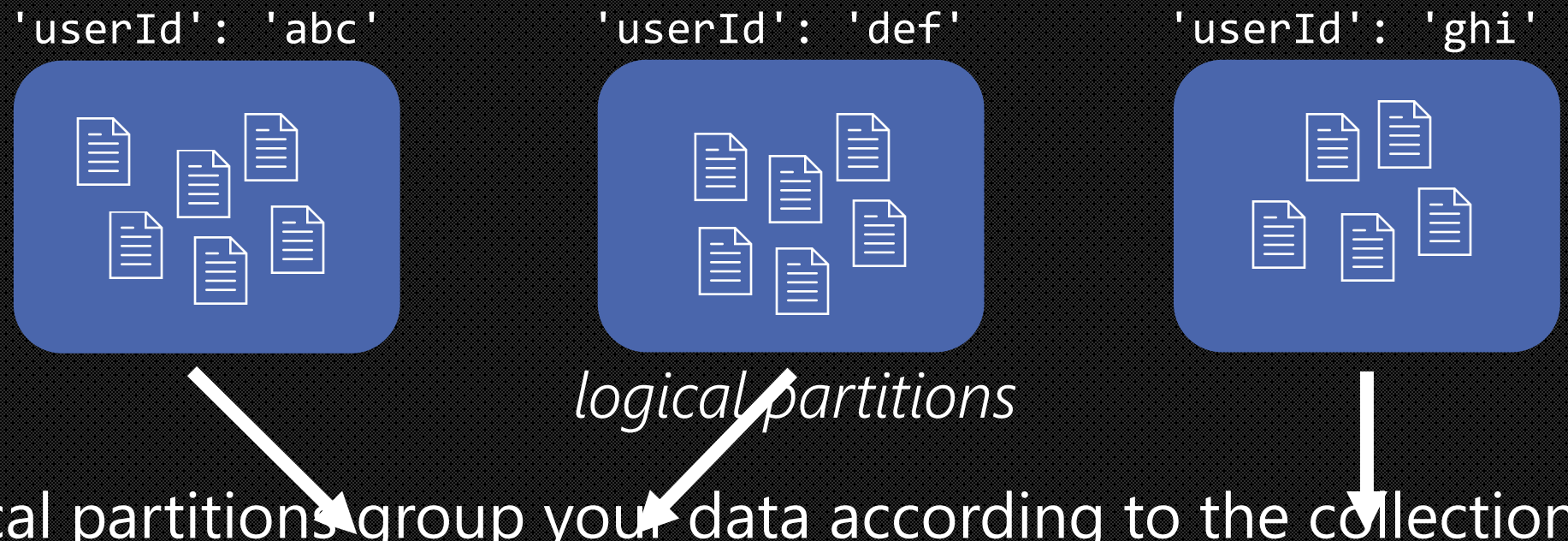
The rules of the game: partitioning

- Cosmos DB achieves horizontal scalability by **partitioning** your data



The rules of the game: partitioning

- Cosmos DB achieves horizontal scalability by **partitioning** your data



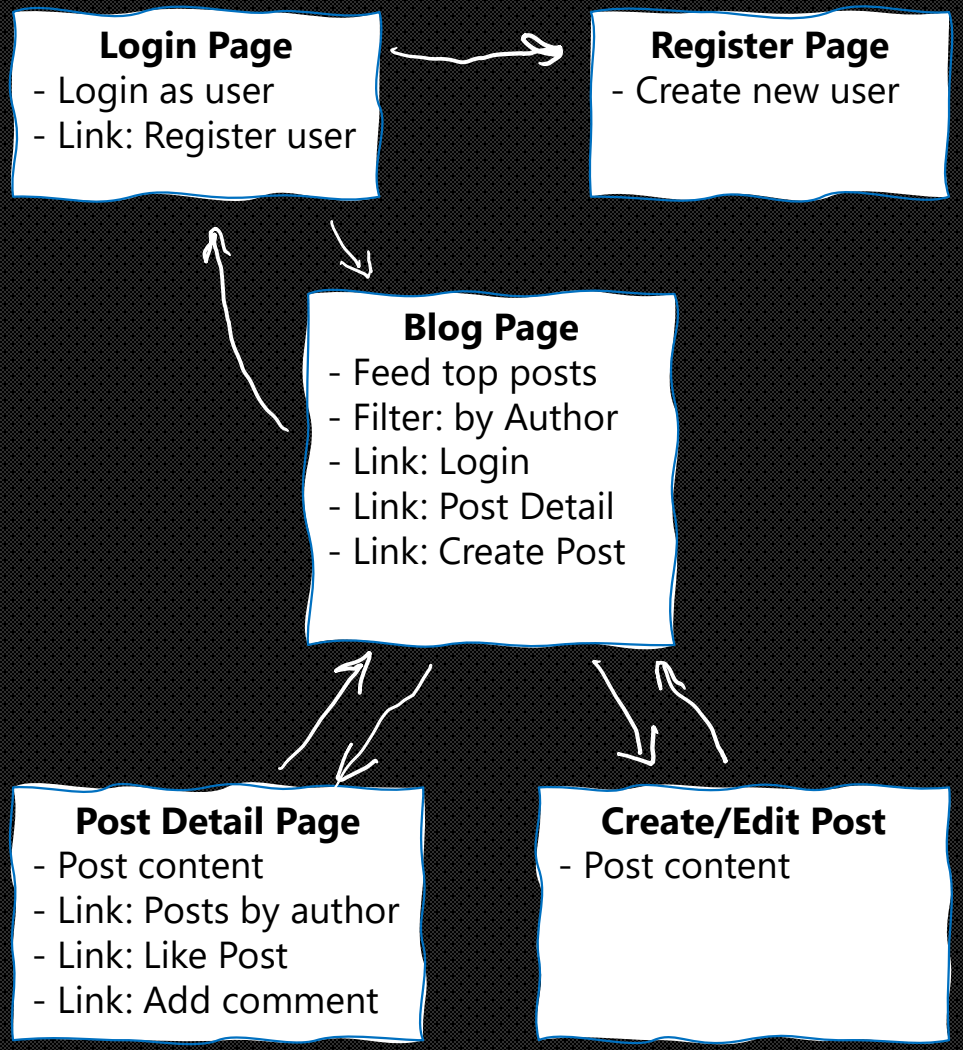
- Logical partitions group your data according to the collection's **partition key**

The rules of the game: partitioning

- A good partition key ensures **well-balanced partitions**
  - Both in terms of storage and in terms of throughput
- Ideally, read queries should get all their results from one partition


```
SELECT * FROM u WHERE u.userId = 'def'
```

Scenario: Rehost Blog in Cosmos



Users can create **posts**. Users can also **like** and add **comments** to posts.

A front page displays a feed of recently created posts. It is possible to list all posts for a particular user. It is also possible to list all comments of a post and all users who have liked a post.

A post is displayed with its author's username and a count of comments and likes. Comments and likes are also displayed with the username of their authors.

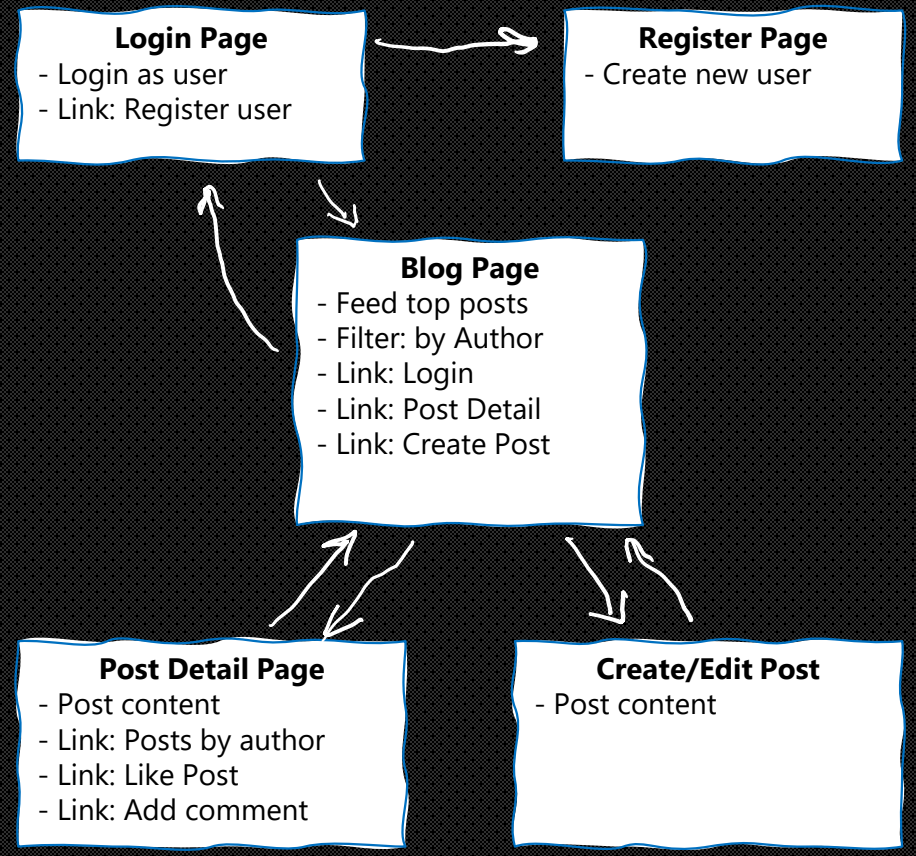
When displayed as lists, posts only present a truncated summary of their content.

Start by identifying the access patterns

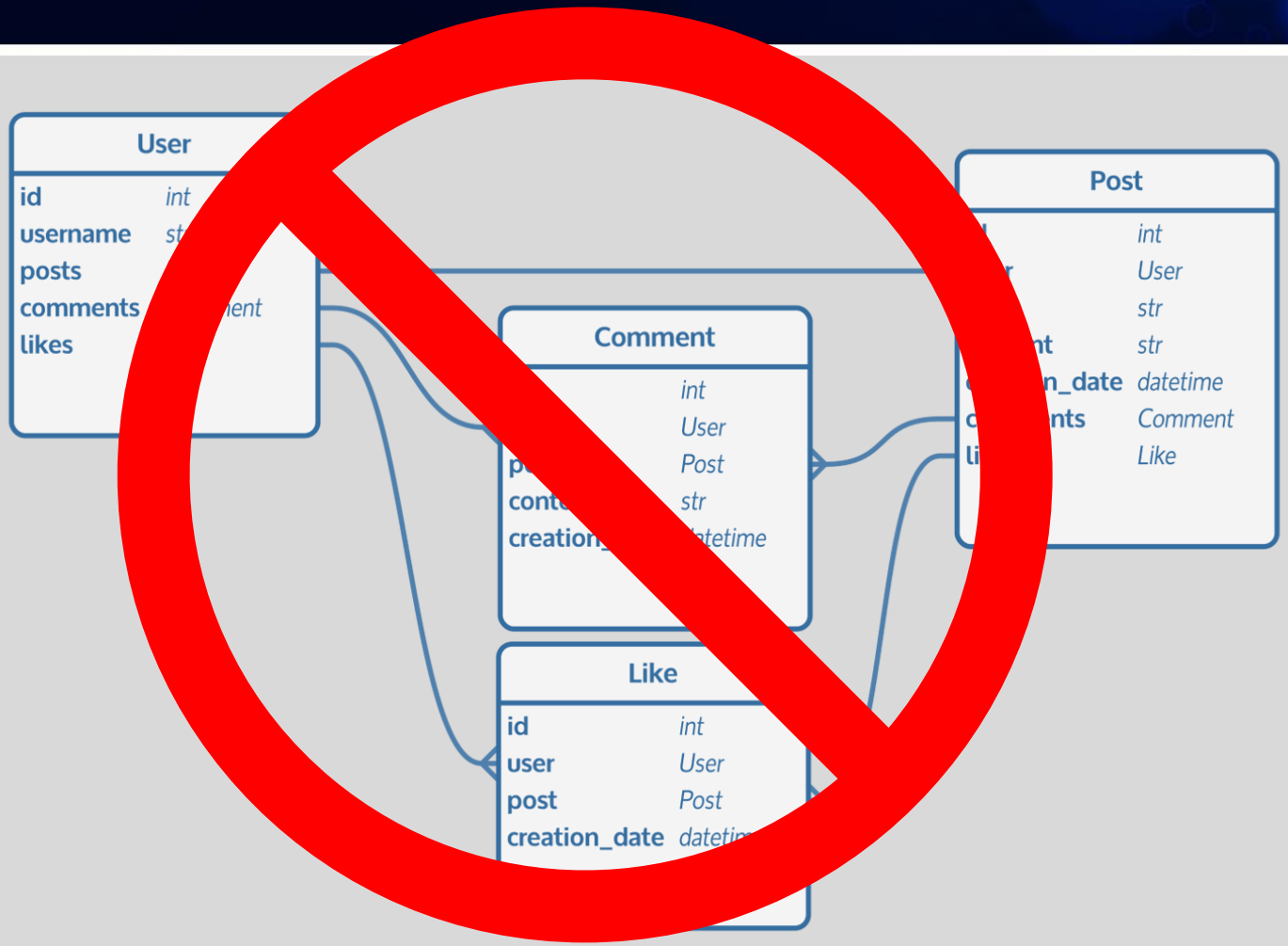
- What are the requests that our design will have to serve?
- To make it easier to follow, categorize them as:
  - **C**ommand (write request)
  - **Q**uery (read-only request)
- Think in terms of data navigation

Start by identifying the access patterns

- **[C1]** Create/edit a user's profile
- **[Q1]** Retrieve a user's profile
- **[C2]** Create/edit a post
- **[Q2]** Retrieve a post
- **[Q3]** List a user's posts in short form
- **[C3]** Create a comment
- **[Q4]** List a post's comments
- **[C4]** Like a post
- **[Q5]** List a post's likes
- **[Q6]** List the *x* most recent posts in short form (feed)



Maybe you instantly think about...

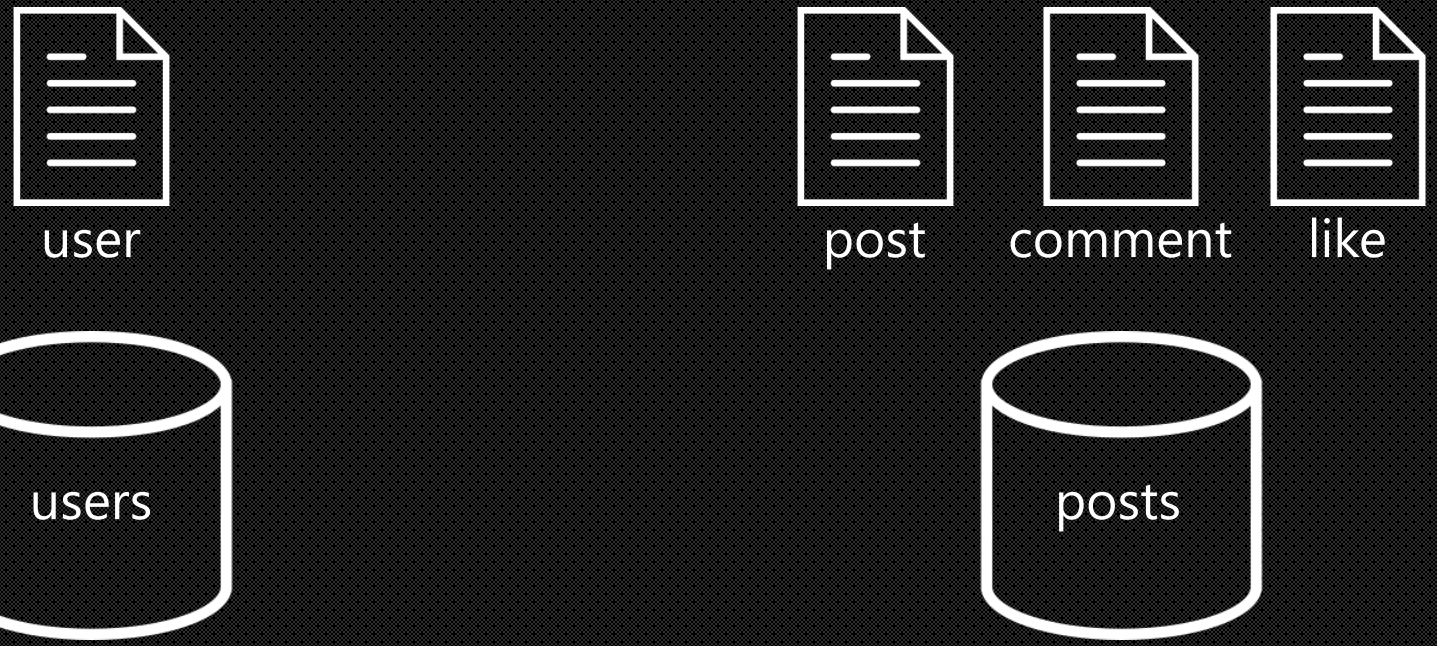


The rules of the game: data model

- In a Cos DB database, store **doc ts** in **ctions**
  - like rows ?

The rules of the game: data model

- In a Cosmos DB database, we store **documents** in **collections**



V1: a first shot

```
{
  "id": "<user-id>",
  "username": "<user-username>"
}
```

```
{
  "id": "<post-id>",
  "userId": "<post-author-id>",
  "title": "<post-title>",
  "content": "<post-content>",
  "creationDate": "<post-creation-date>"
}
```

```
{
  "id": "<comment-id>",
  "postId": "<post-id>",
  "userId": "<comment-author-id>",
  "content": "<comment-content>",
  "creationDate": "<comment-creation-date>"
}
```

```
{
  "id": "<like-id>",
  "postId": "<post-id>",
  "userId": "<liker-id>",
  "creationDate": "<like-creation-date>"
}
```

To embed or to reference?

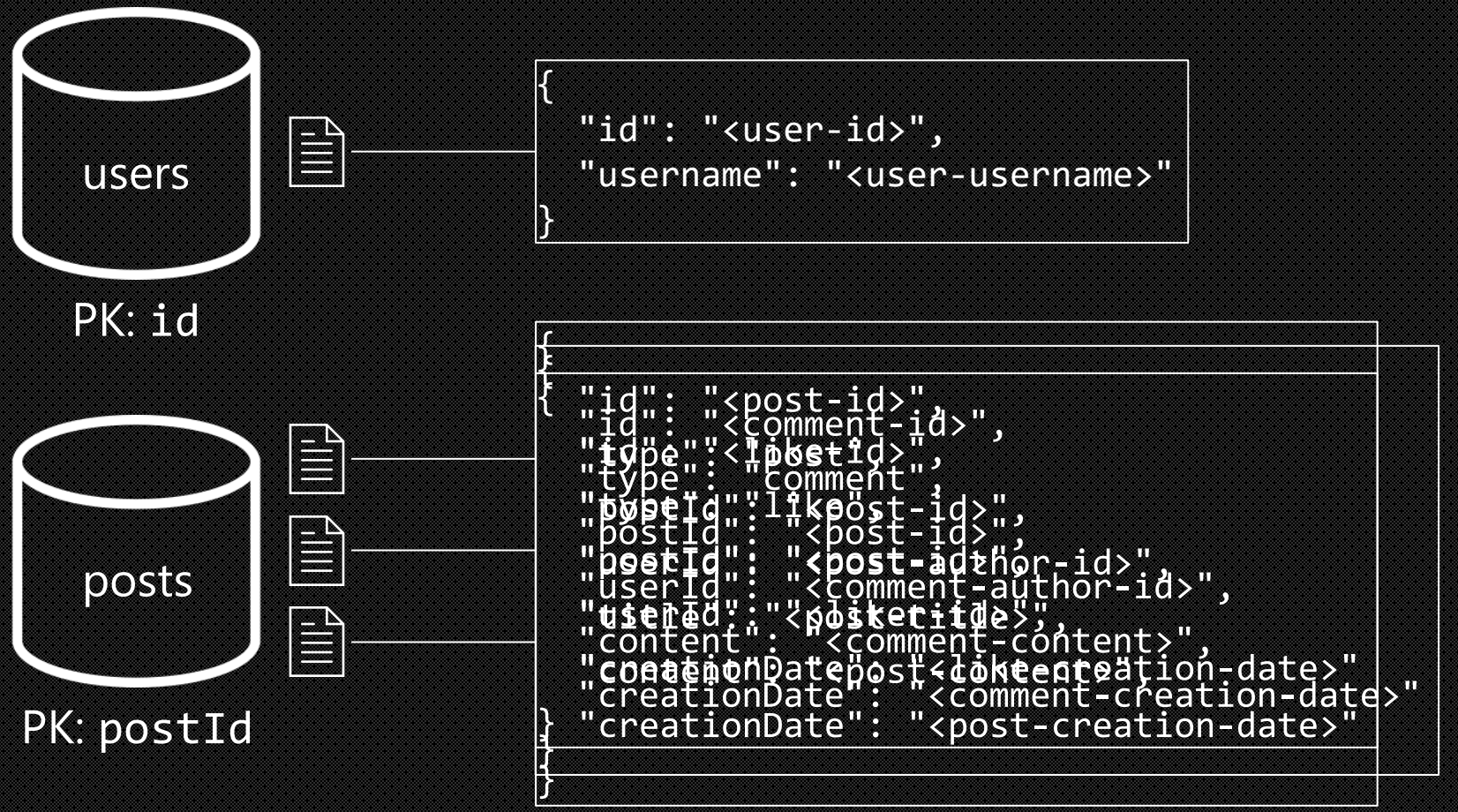
Embed when:

- 1:1
- 1:few
- Items queried together
- Items updated together

Reference when:

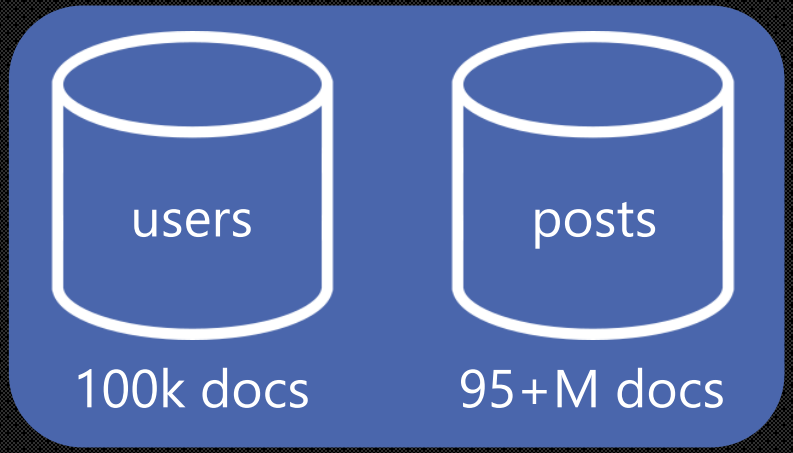
- 1:many (unbounded)
- Many:many
- Items updated independently

V1: a first shot



How well does our model perform?

- Benchmarked against a dummy dataset
  - 100,000 users
  - 5 – 50 posts / user
  - 0 – 25 comments / post
  - 0 – 100 likes / post



100 GB

Create/edit a user's profile

C1

Q1

C2

Q2

Q3

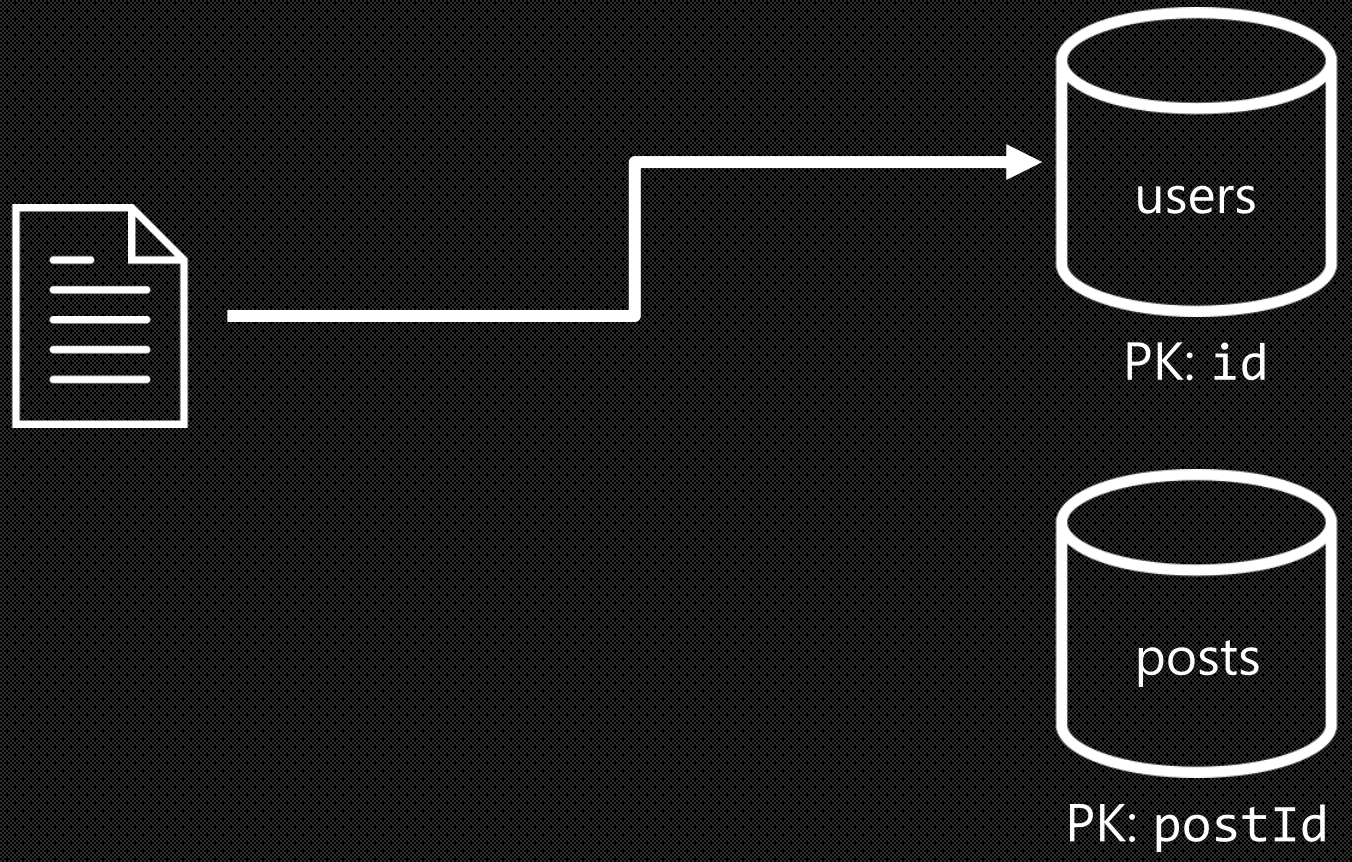
C3

Q4

C4

Q5

Q6



7 ms / 5.71 RU

Retrieve a user's profile

C1 ✓

Q1 ✓

C2

Q2

Q3

C3

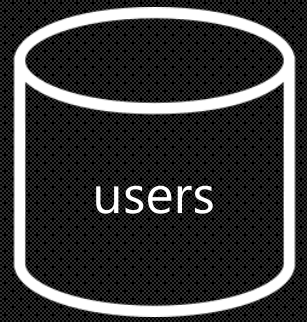
Q4

C4

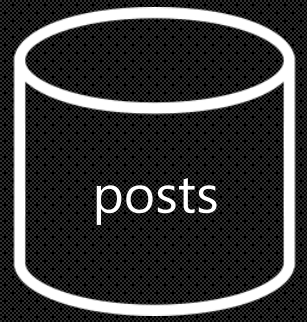
Q5

Q6

```
SELECT * FROM u WHERE u.id = '<user-id>'
```



PK: `id`

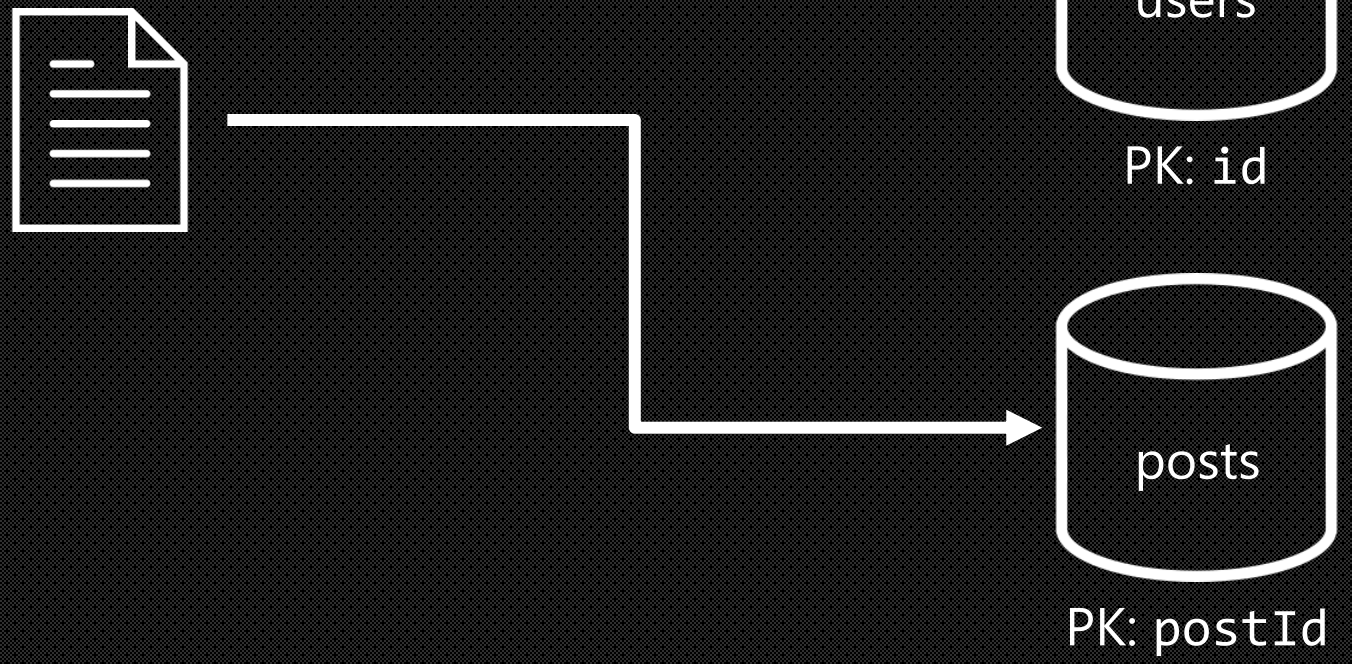


PK: `postId`

3 ms / 2.90 RU

Create/edit a post

C1 ✓
Q1 ✓
C2 ✓
Q2
Q3
C3
Q4
C4
Q5
Q6



9 ms / 8.76 RU

Retrieve a post

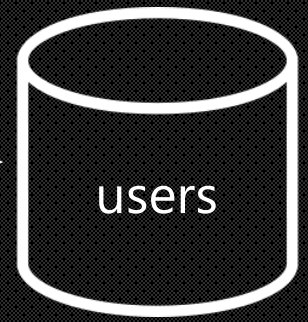
C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3
C3
Q4
C4
Q5
Q6

```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

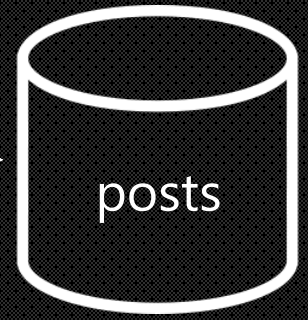
```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'post'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



PK: id



PK: postId

9 ms / 19.54 RU

List a user's posts in short form

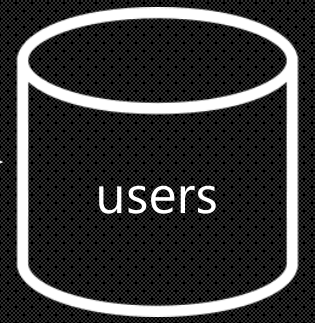
C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3 ⚠
C3
Q4
C4
Q5
Q6

```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

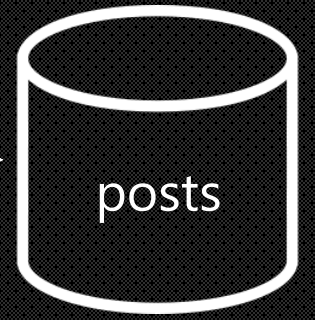
```
SELECT * FROM p
WHERE p.userId = '<author-id>'
AND p.type = 'post'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



PK: id

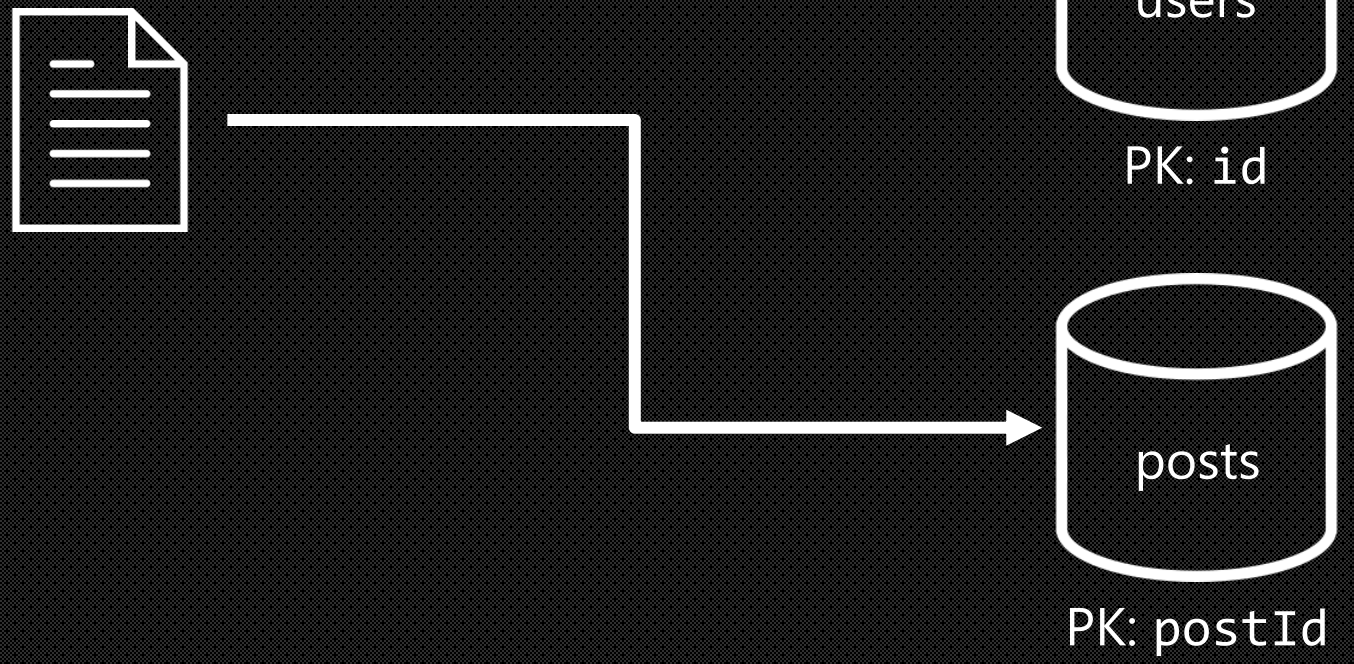


PK: postId

130 ms / 619.41 RU

Create a comment

C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3 ⚠
C3 ✓
Q4
C4
Q5
Q6



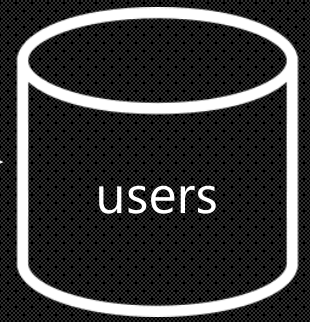
7 ms / 8.57 RU

List a post's comments

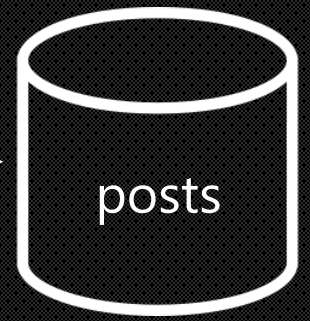
C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3 ⚠
C3 ✓
Q4 ⚠
C4
Q5
Q6

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT u.username FROM u
WHERE u.id = '<comment-author-id>'
```



PK: id

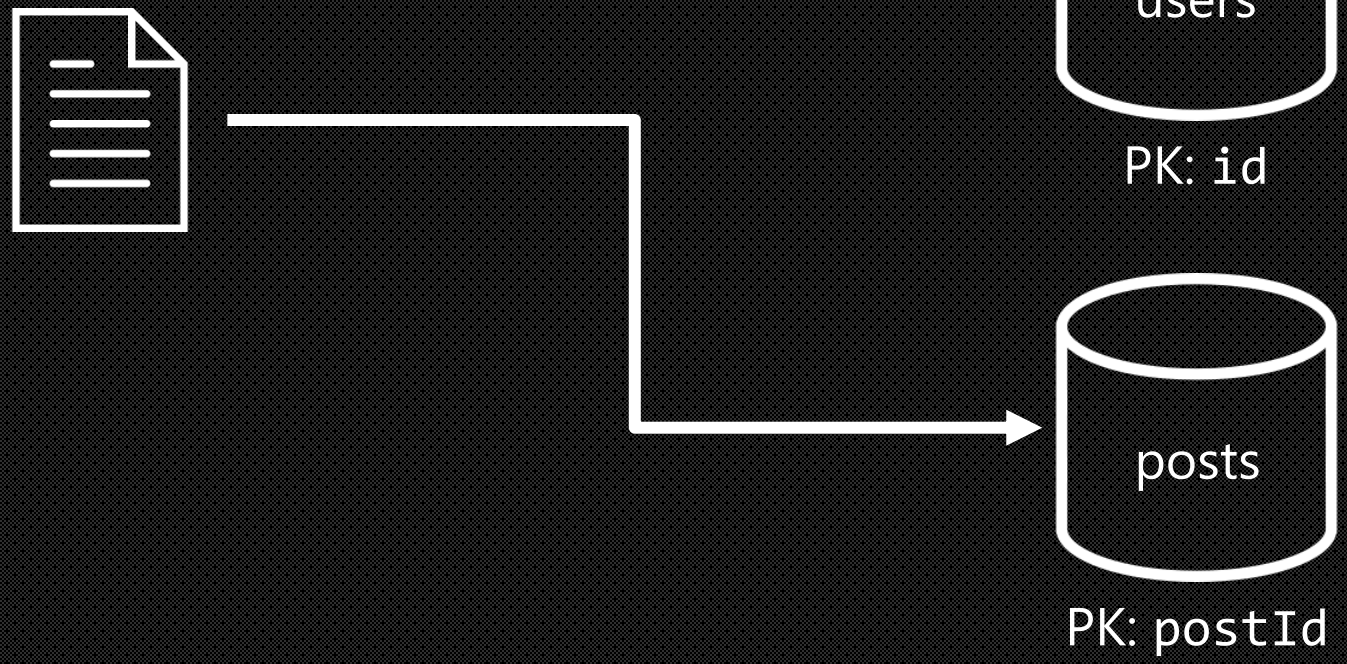


PK: postId

23 ms / 27.72 RU

Like a post

C1
Q1
C2
Q2
Q3
C3
Q4
C4
Q5
Q6



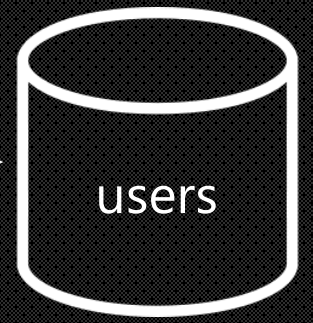
6 ms / 7.05 RU

List a post's likes

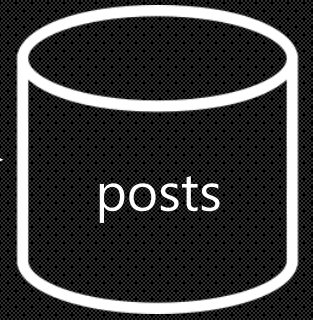
C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3 ⚠
C3 ✓
Q4 ⚠
C4 ✓
Q5 ⚠
Q6

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```

```
SELECT u.username FROM u
WHERE u.id = '<liker-id>'
```



PK: id



PK: postId

59 ms / 58.92 RU

List the *x* most recent posts in short form (feed)

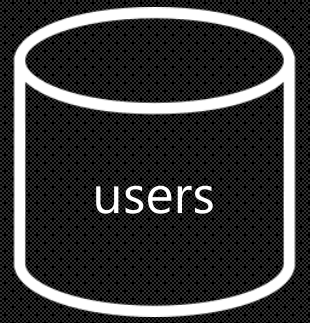
C1 ✓
Q1 ✓
C2 ✓
Q2 ⚠
Q3 ⚠
C3 ✓
Q4 ⚠
C4 ✓
Q5 ⚠
Q6 ⚠

```
SELECT TOP <x> * FROM p
WHERE p.type = 'post'
ORDER BY p.creationDate DESC
```

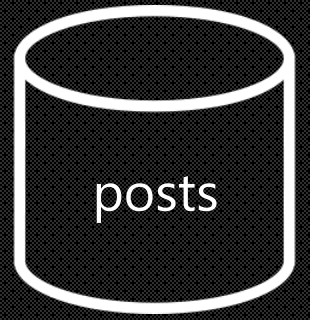
```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



PK: id



PK: postId

306 ms / 2063.54 RU

Cost in RU's

C1	Create/edit a user's profile	5.71 RU
Q1	Retrieve a user's profile	2.90 RU
C2	Create/edit a post	8.76 RU
Q2	Retrieve a post	19.54 RU
Q3	List a user's post in short form	619.41 RU
C3	Create a comment	8.57 RU
Q4	List a post's comments	27.72 RU
C4	Like a post	7.05 RU
Q5	List a post's likes	58.92 RU
Q6	List the x most recent posts in short form (feed)	2,063.54 RU

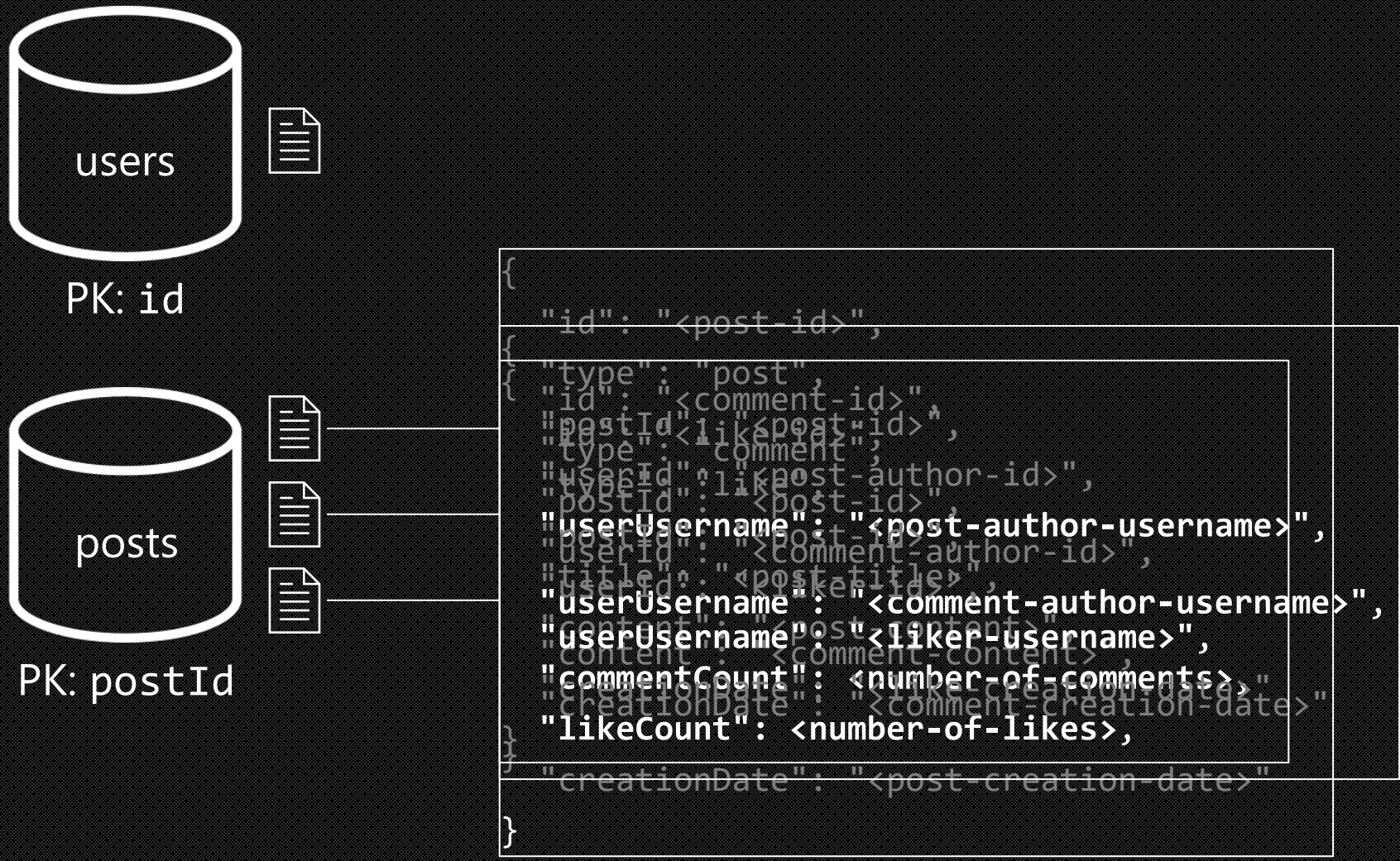
What we'll cover today

Why Cosmos
Provisioning Cosmos
Working with Data
Data Modeling Strategies
Performance and tuning

V1: performance issues

- Having to issue multiple queries to handle a single request
- Issuing queries that don't filter on the partition key, leading to a partition scan

V2: introducing denormalization



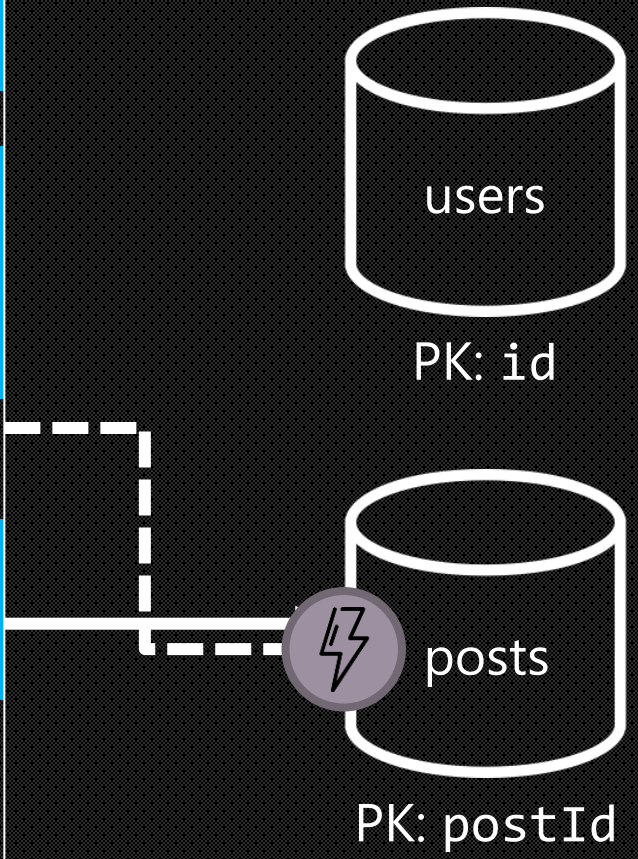
Denormalizing within the same logical partition with stored procedures

- Written in Javascript
- Scoped to a single logical partition
- Executed as atomic transactions

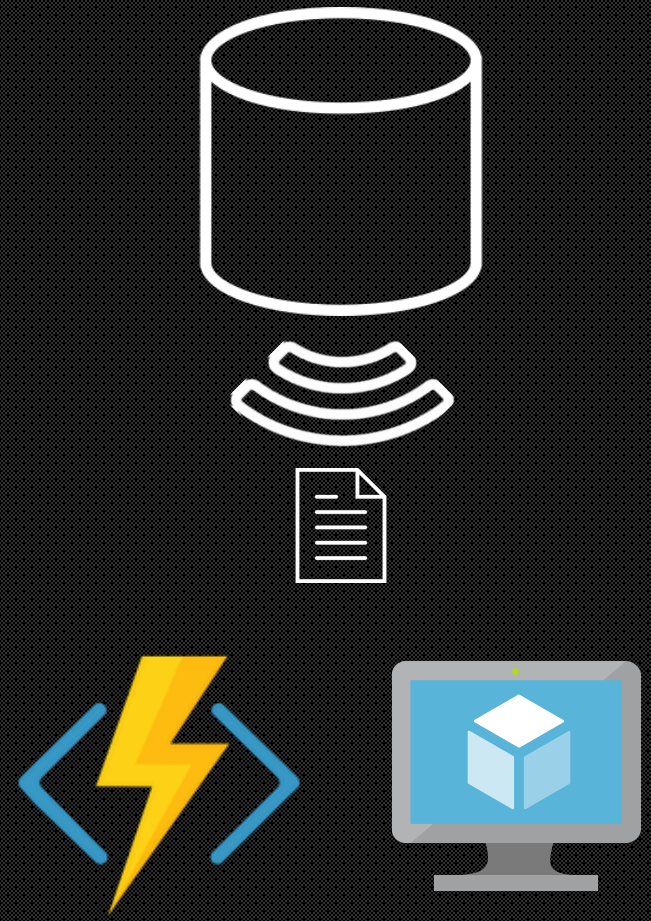
Create a comment

C1
Q1
C2
Q2
Q3
C3
Q4
C4
Q5
Q6

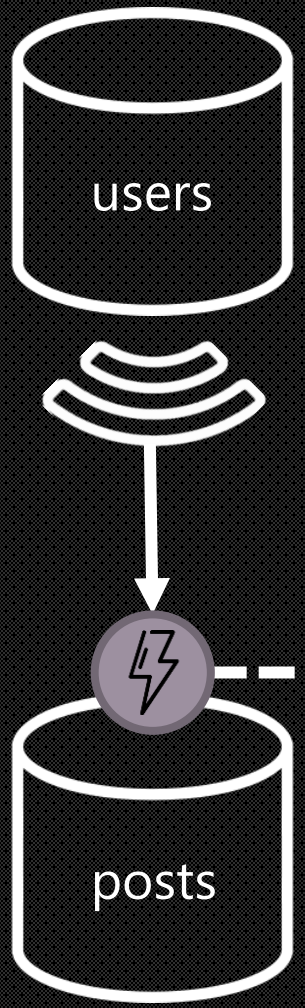
```
function createComment(postId, comment) {
  var collection = getContext().getCollection();
  collection.readDocument(
    `${collection.getAltLink()}/docs/${postId}`,
    function (err, post) {
      post.commentCount++;
      collection.replaceDocument(
        post._self,
        post,
        function (err) {
          comment.postId = postId;
          collection.createDocument(
            collection.getSelfLink(),
            comment
          );
        }
      );
    });
}
```



Denormalizing across logical partitions and containers with the change feed



Denormalizing usernames into posts




```
function updateUsername(userId, username) {
  var collection = getContext().getCollection();
  collection.queryDocuments(
    collection.getSelfLink(),
    `SELECT * FROM p WHERE p.userId = '${userId}'`,
    function (err, results) {
      for (var i in results) {
        var doc = results[i];
        doc.userUsername = username;
        collection.replaceDocument(
          collection.getSelfLink(),
          doc);
      }
    });
}
```

Retrieve a post

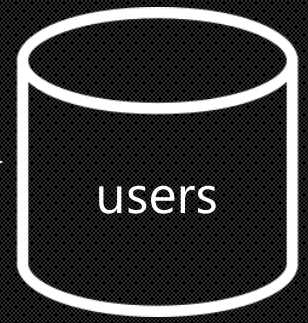
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ⚠
C3 ✓
Q4 ⚠
C4 ✓
Q5 ⚠
Q6 ⚠

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'post'
```

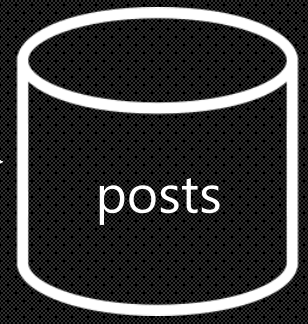
```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



PK: id



PK: postId

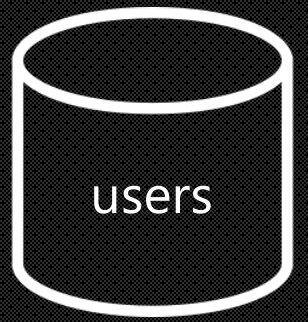
3 ms / 5.40 RU

List a post's comments

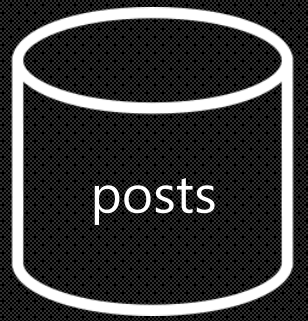
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ⚠
C3 ✓
Q4 ✓
C4 ✓
Q5 ⚠
Q6 ⚠

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT u.username FROM u
WHERE u.id = '<comment-author-id>'
```



PK: id



PK: postId

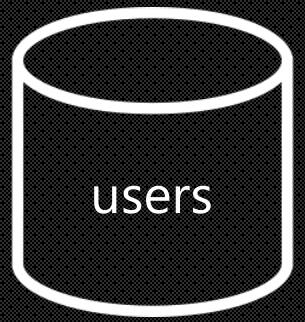
4 ms / 7.72 RU

List a post's likes

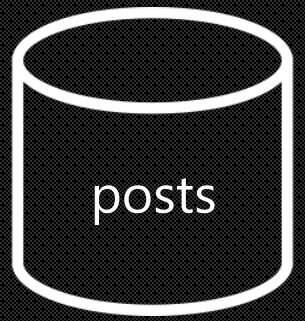
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ⚠
C3 ✓
Q4 ✓
C4 ✓
Q5 ✓
Q6 ⚠

```
SELECT * FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```

```
SELECT u.username FROM u
WHERE u.id = '<liker-id>'
```



PK: id



PK: postId

4 ms / 8.92 RU

List a user's posts in short form

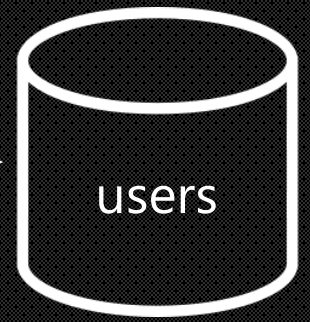
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ⚠
C3 ✓
Q4 ✓
C4 ✓
Q5 ✓
Q6 ⚠

```
SELECT * FROM p
WHERE p.userId = '<author-id>'
AND p.type = 'post'
```

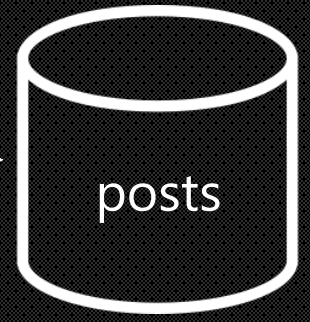
```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



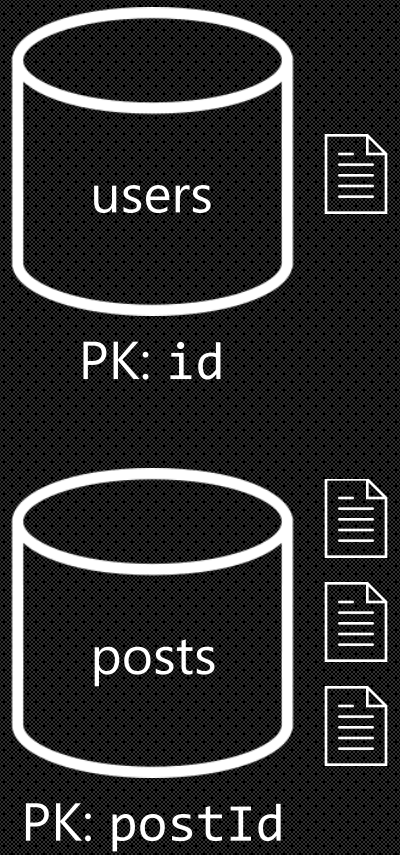
PK: id



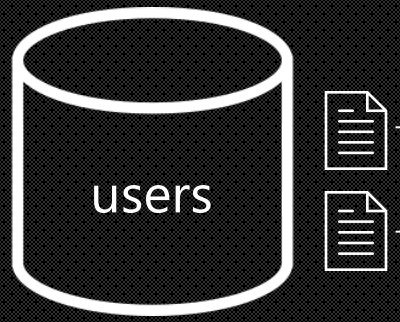
PK: postId

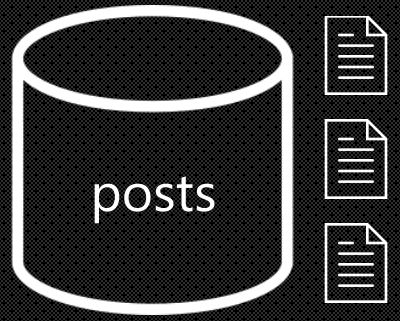
83 ms / 532.33 RU

V3: denormalizing entire documents



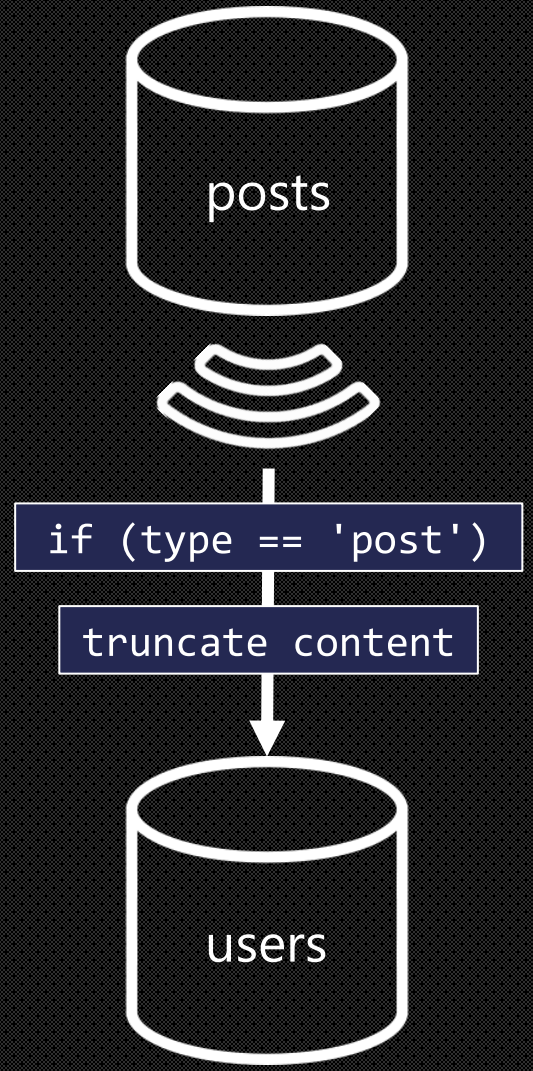
V3: denormalizing entire documents






```
{
  "id": "<post-id>",
  "type": "post",
  "userId": "<post-author-id>",
  "username": "<post-author-username>",
  "title": "<post-title>",
  "content": "<post-content>",
  "commentCount": <number-of-comments>,
  "likeCount": <number-of-likes>,
  "creationDate": "<post-creation-date>"
}
```

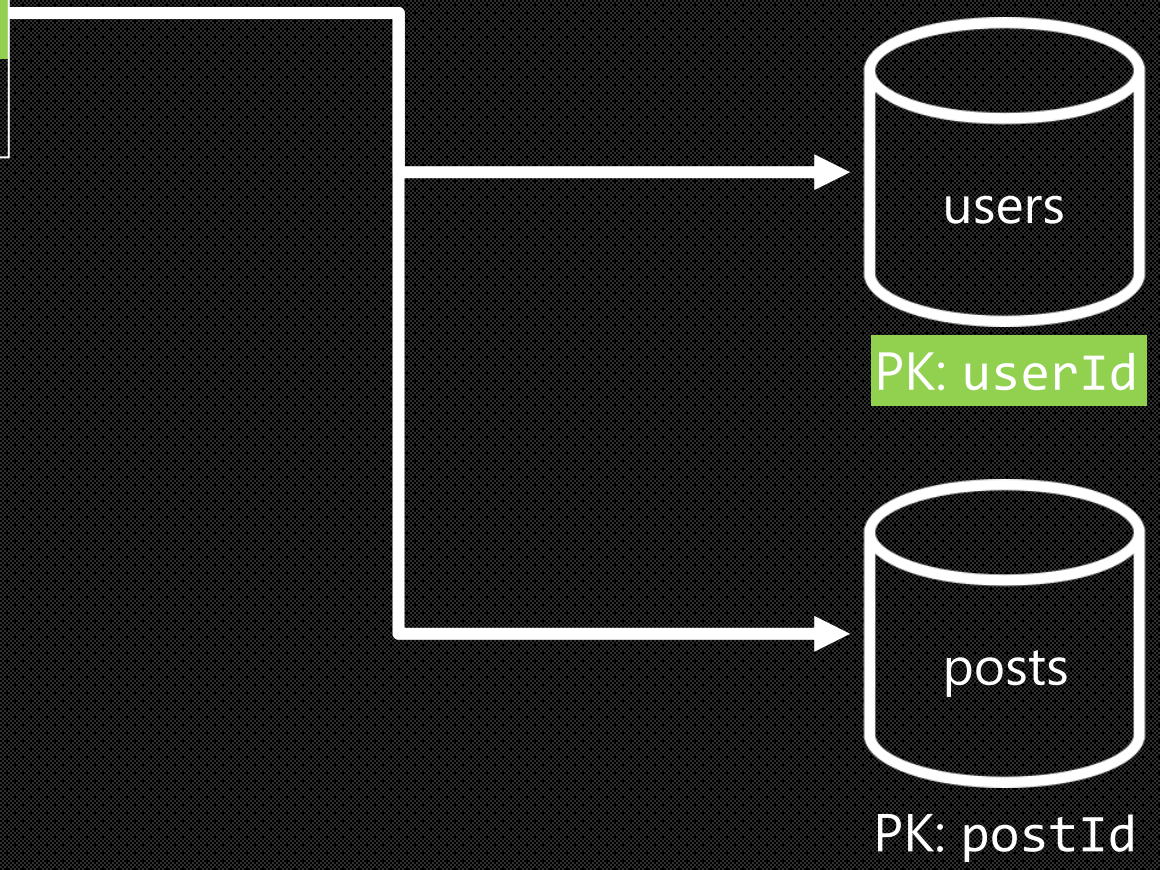
Denormalizing documents with the change feed



List a user's posts in short form

C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ✓ ⚠
C3 ✓
Q4 ✓
C4 ✓
Q5 ✓
Q6 ⚠

```
SELECT * FROM p
WHERE p.userId = '<author-id>'
AND p.type = 'post'
```



4 ms / 6.46 RU

List the *x* most recent posts in short form (feed)

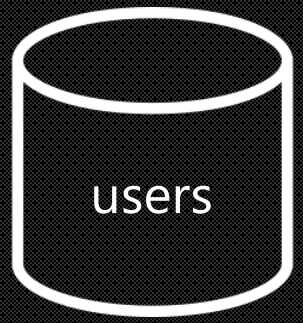
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ✓
C3 ✓
Q4 ✓
C4 ✓
Q5 ✓
Q6 ⚠

```
SELECT TOP <x> * FROM p
WHERE p.type = 'post'
ORDER BY p.creationDate DESC
```

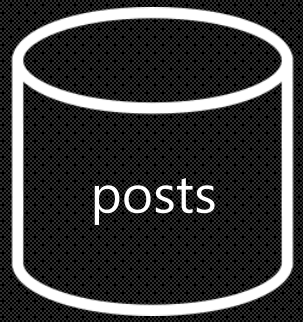
```
SELECT u.username FROM u
WHERE u.id = '<post-author-id>'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'comment'
```

```
SELECT COUNT(1) FROM p
WHERE p.postId = '<post-id>'
AND p.type = 'like'
```



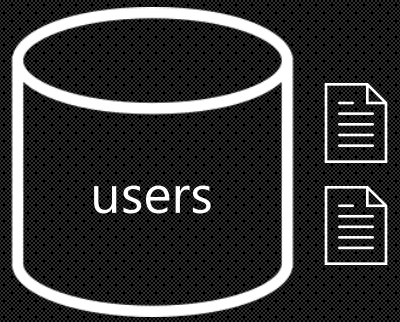
PK: id



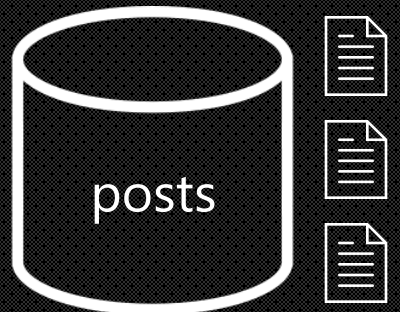
PK: postId

9 ms / 16.97 RU

V3: denormalizing entire documents

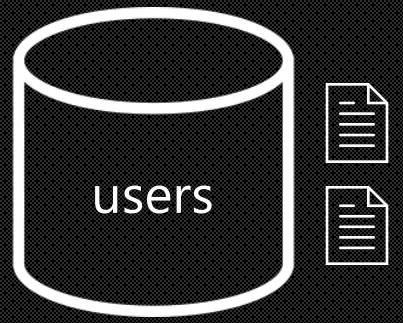


PK: userId

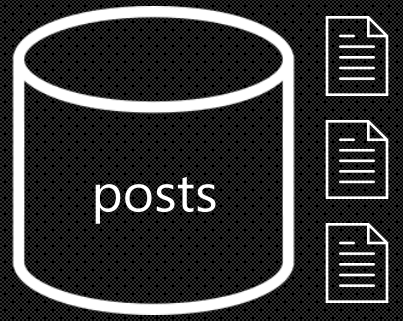


PK: `postId`

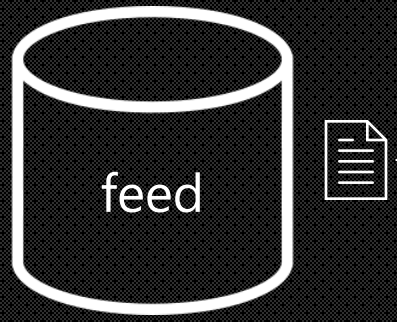
V3: denormalizing entire documents



PK: userId



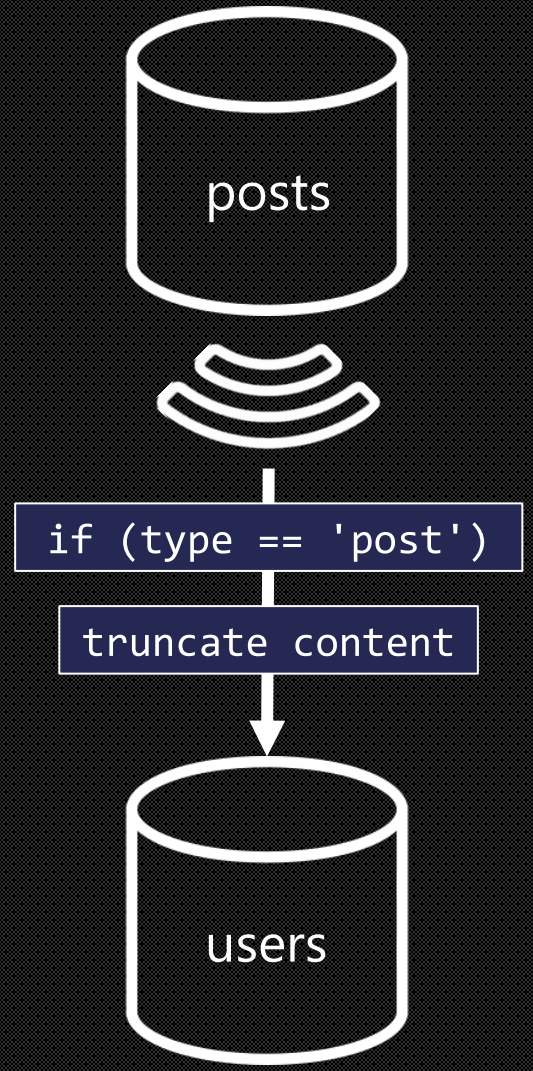
PK: postId



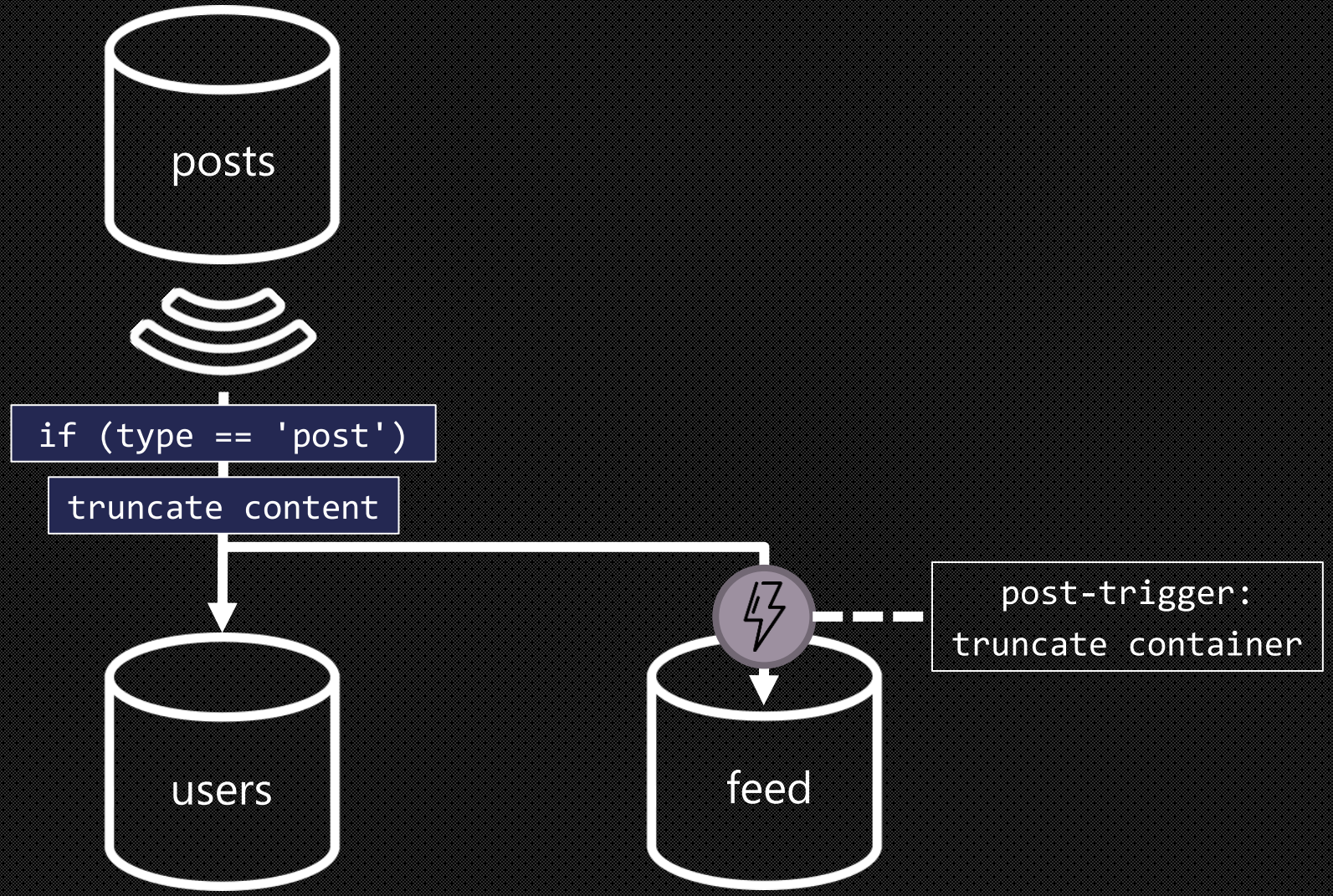
PK: type

```
{
  "id": "<post-id>",
  "type": "post",
  "userId": "<post-author-id>",
  "userUsername": "<post-author-username>",
  "title": "<post-title>",
  "content": "<post-content>",
  "commentCount": <number-of-comments>,
  "likeCount": <number-of-likes>,
  "creationDate": "<post-creation-date>"
}
```

Denormalizing documents with the change feed



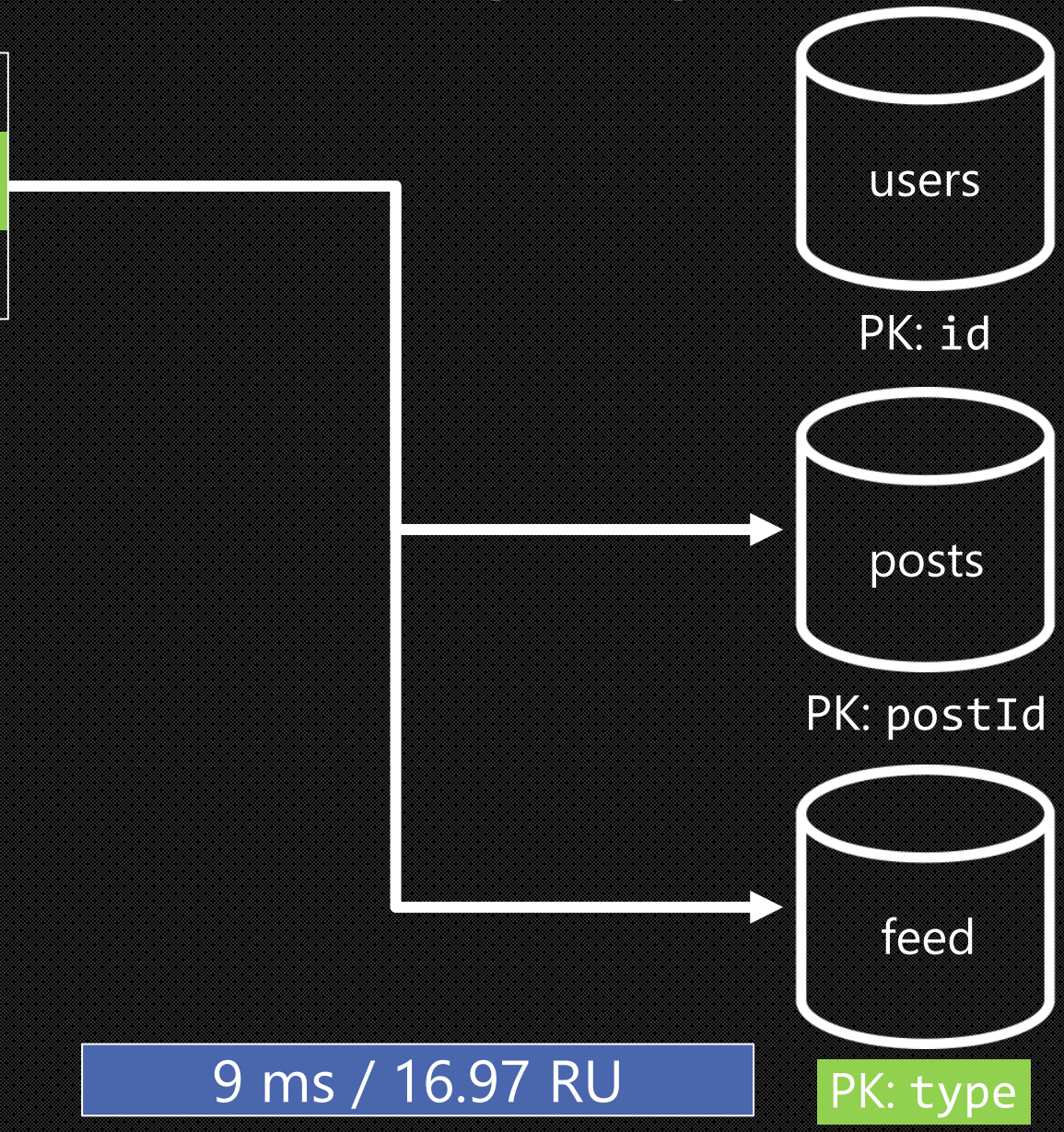
Denormalizing documents with the change feed



List the *x* most recent posts in short form (feed)

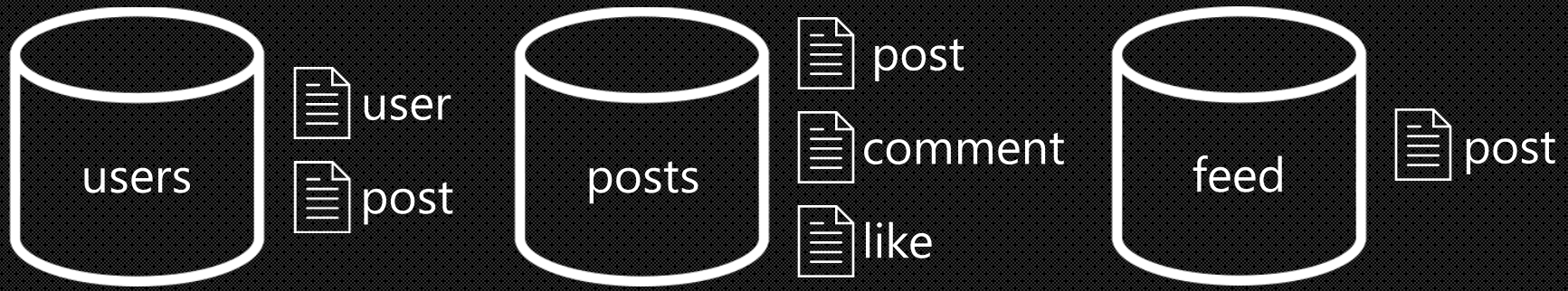
C1 ✓
Q1 ✓
C2 ✓
Q2 ✓
Q3 ✓
C3 ✓
Q4 ✓
C4 ✓
Q5 ✓
Q6 ✓

```
SELECT TOP <x> * FROM p
WHERE p.type = 'post'
ORDER BY p.creationDate DESC
```

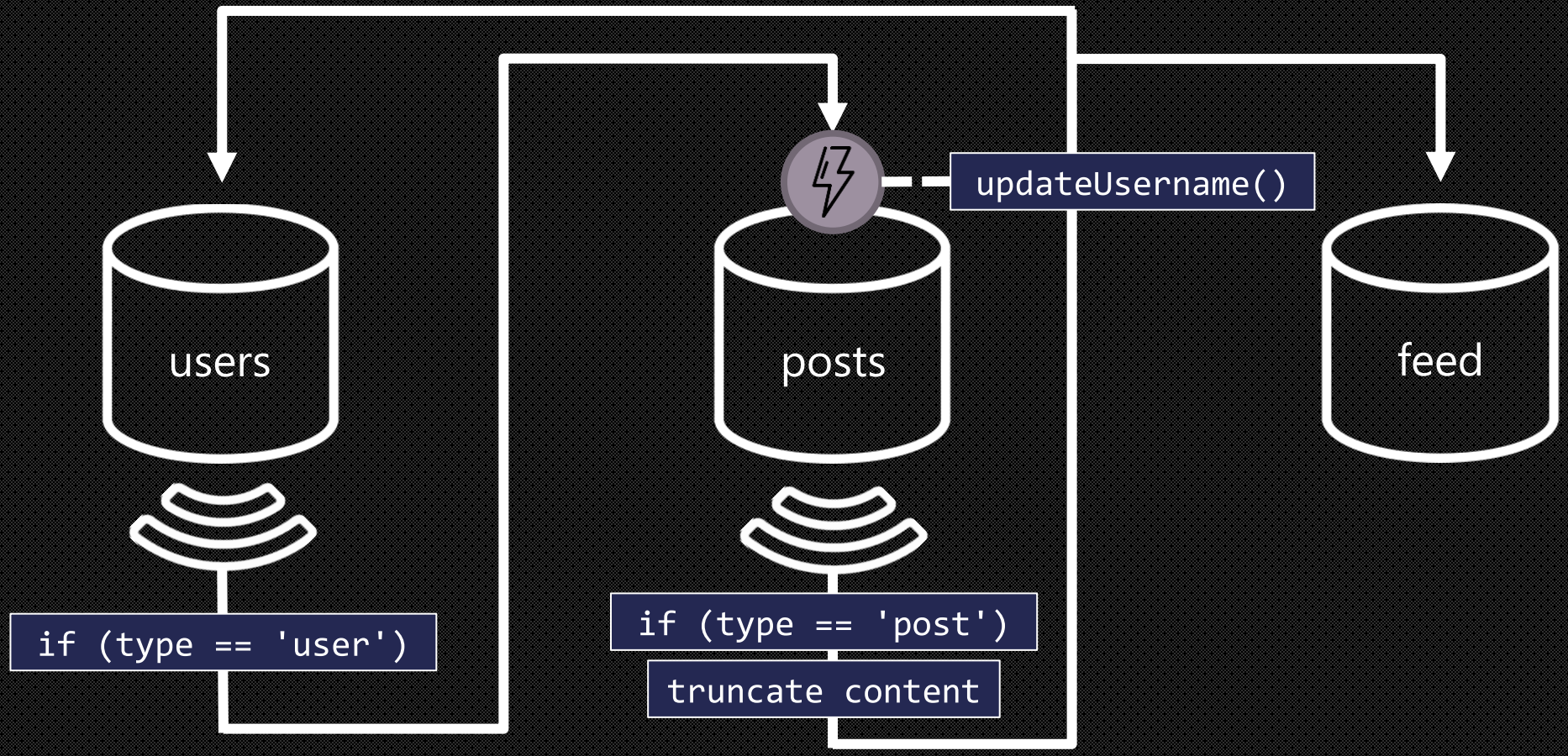


9 ms / 16.97 RU

Our final design



Our final design



Denormalization can be added later

- V1 was not that bad!
- You can add denormalization later
  - The **change feed is persistent**
  - Run a one-time catch-up operation
- **Monitor** your database performance and setup **alerts**

	V1
C1	7 ms / 5.71 RU
Q1	3 ms / 2.90 RU
C2	9 ms / 8.76 RU
Q2	9 ms / 19.54 RU
Q3	130 ms / 619.41 RU
C3	7 ms / 8.57 RU
Q4	23 ms / 27.72 RU
C4	6 ms / 7.05 RU
Q5	59 ms / 58.92 RU
Q6	306 ms / 2063.54 RU

An overall look at performance

	V1	V2	V3
C1	7 ms / 5.71 RU	7 ms / 5.71 RU	7 ms / 5.71 RU
Q1	3 ms / 2.90 RU	3 ms / 2.90 RU	3 ms / 2.90 RU
C2	9 ms / 8.76 RU	9 ms / 8.76 RU	9 ms / 8.76 RU
Q2	9 ms / 19.54 RU	3 ms / 5.11 RU	3 ms / 5.11 RU
Q3	130 ms / 619.41 RU	28 ms / 201.54 RU	4 ms / 6.46 RU
C3	7 ms / 8.57 RU	7 ms / 15.27 RU	7 ms / 15.27 RU
Q4	23 ms / 27.72 RU	4 ms / 7.72 RU	4 ms / 7.72 RU
C4	6 ms / 7.05 RU	7 ms / 14.67 RU	7 ms / 14.67 RU
Q5	59 ms / 58.92 RU	4 ms / 8.92 RU	4 ms / 8.92 RU
Q6	306 ms / 2063.54 RU	83 ms / 532.33 RU	9 ms / 16.97 RU

What about the cost of denormalization?

- We optimized for a read-heavy scenario
  - Reads >> writes
- Q6 went from 2,000+ to 17 RUs (120x optimization)
  - Saved 1920 RUs by denormalizing posts at ~10 RUs / document
- All depends on your specific scenario and access patterns

What else do you get?

- Multi-model APIs
- Flexibility of the change feed
- Turnkey global distribution
- Industry-leading SLA

Most importantly: what's our scalability?

- V3: limitless!
- Performance will ***not*** depend on
  - number of documents
  - number of users
  - number of posts
- Whether you have 10 or 10M users, query latency will always be **the same**

	V3
C1	7 ms / 5.71 RU
Q1	3 ms / 2.90 RU
C2	9 ms / 8.76 RU
Q2	3 ms / 5.11 RU
Q3	4 ms / 6.46 RU
C3	7 ms / 14.67 RU
Q4	4 ms / 7.72 RU
C4	7 ms / 14.67 RU
Q5	4 ms / 8.92 RU
Q6	9 ms / 16.97 RU

What we'll cover today

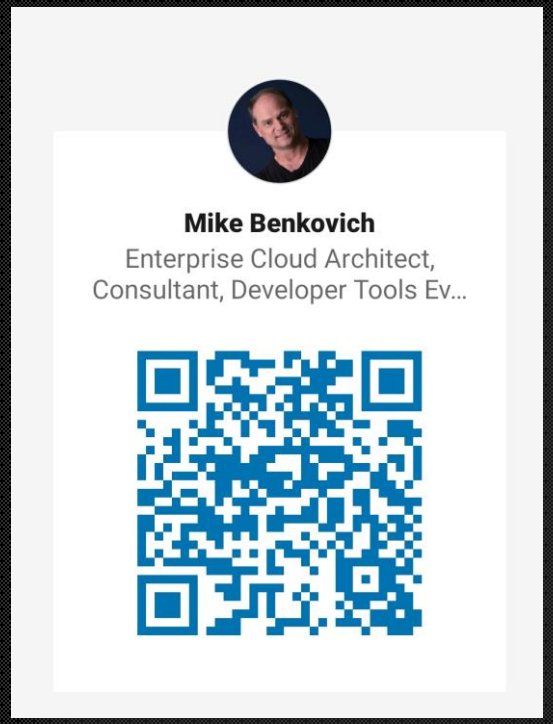
Why Cosmos
Provisioning Cosmos
Working with Data
Data Modeling Strategies
Performance and tuning

Thanks to the following

- **Thomas Weiss** presentation on Cosmos Performance in 2019
- **Alpaqa Studio**
  https://alpaqastudio.com
- **Jeff Widmer** github code example of scenario
  https://github.com/jwidmer/AzureCosmosDbBlogExample
- **Jane Chen** - Cosmos Jupyter Notebooks
- **Mark Brown's** Cosmos TV
  https://gotcosmos.com
- **James Serra**'s Blog
  https://www.jamesserra.com

Mike Benkovich

- **Developer**
- Cloud **Architect** & **Consultant**
- Live in **Minneapolis**
- Founder of **Imagine Technologies**, Inc.
- Developing Courses for **LinkedIn** Learning
- Blog www.**benkoTIPS**.com
- Follow **@mbenko** on **Twitter**
- Send me **Feedback**! mike@benkotips.com



Going further

Questions? I'm mike@benko.com - www.benko**TIPS**.com

- aka.ms/PracticalCosmosDB
- cosmosdb.com
- https://github.com/jwidmer/AzureCosmosDbBlogExample