BOARDROOM · MAY 19 · 17:15 – 17:40

# Understanding your graph, one hash at a time.

How federation's entity keys become a measurement instrument.

**Jens Neuse**
Founder & CEO · WunderGraph

# What is this?

employees.graphql

```graphql
type Query {
  employee(id: Int!): Employee
}

type Employee @key(fields: "id") {
  id: Int!
  name: String!
}
```

# Three words: subgraph, entity, key.

That is the whole vocabulary of GraphQL federation.

01 · subgraph

## Subgraph

A service that owns part of the graph. The team that ships it owns its schema, its resolvers, its database.

02 · entity

## Entity

A type resolvable by more than one service. Declared with `@key` so the router can stitch it across subgraphs.

03 · key

## Key

`@key(fields: "id")` — a stable, portable address for one object across the whole distributed graph.

# A key is an address.

Give your data addresses and you can build anything on top of them.

COSMO STREAMS

**The key routes an event to an entity.**

streams.graphql

```
type Employee @key(fields: "id", resolvable: false) {
  id: Int!
}

type Subscription {
  employeeUpdates(id: Int!):
    Employee!
    @edfs__natsSubscribe(
      subjects: ["employees.{{ args.id }}"]
    )
}
```

AUTHORIZATION

**The key names the instance an access rule applies to.**

authz.graphql

```
type Employee @key(fields: "id") {
  id: Int!
  salary: Int!
    @requiresScopes(scopes: [["hr:read"]])
  review: String
    @authenticated
}
```

# Another thing you can build on a key: a cache.

If an entity is expensive to resolve, cache it.

**But caching anything distributed needs one hard thing first —**
**a good cache key.**

# *You already designed your cache key.*
# *You called it* @key.

THE ROOM CONVERGES ON

- Stable across services
- Portable, no shared runtime
- Uniquely identifies one instance
- Same input, same key, forever

THE ACTUAL CACHE KEY

```
{ "__typename": "Employee",
  "key": { "id": 1 } }
```

The federation entity key ***is*** the cache key.

# *The schema tells you what is possible.*
# *It never tells you what is happening.*

Slap a cache on — it works in the demo. Then the invisible problems start.

P · 01 · header forwarding

## Header forwarding

Forward `Authorization` into the key and you fragment the cache per user. Did the header actually change the response?

P · 02 · invalidation

## Invalidation

A mutation updated the entity — did the cache clear, or do you serve stale reads until the TTL expires?

P · 03 · ttl

## TTL

Too short and you never get a hit. Too long and you serve stale data. What is right?

P · 04 · cardinality

## Key cardinality

50 distinct keys and 5 million behave nothing alike. An over-keyed cache never warms up.

# *Turning the cache on is one directive.*
# *Knowing what to put in it is the rest of this talk.*

## MINIMAL FORM

employees.graphql · minimal

```graphql
type Employee @key(fields: "id")
  @openfed__entityCache(maxAge: 60) {
  id: Int!
  name: String!
}

type Mutation {
  updateEmployee(input: EmployeeInput!):
    Employee!
    @openfed__cacheInvalidate
}
```

## FULL SURFACE · FOUR ARGUMENTS, FOUR PROBLEMS

employees.graphql · full

```graphql
@openfed__entityCache(
  maxAge: 60,
  includeHeaders: true,
  shadowMode: true
)
```

| | |
|---|---|
| `maxAge: 60` | why not 600? why not 5? nothing told you. |
| `includeHeaders` | fragment per user, or serve the wrong response — you are guessing. |
| `@cacheInvalidate` | wired per mutation by hand. Miss one and it leaks stale reads. |
| `shadowMode` | the honest way to find out. Returns on slide 14. |

# Operating an entity cache collapses to four questions.

The schema is silent on all four. You have to observe.

Q · 01

CACHEABLE?

## Is this entity cacheable?

Reads outnumber writes by how much? Are there errors? Is there any hit-rate to extract at all?

Q · 02

UPDATED?

## How often is it updated?

Write frequency caps your TTL. A row mutated every second cannot be cached for a minute.

Q · 03

ACCESSED?

## How often is it accessed?

Read frequency is the upside. 40k reads/day on 10 keys is a different conversation than 40 reads on 4 million.

Q · 04

KEY DISTRIBUTION?

## What is its key distribution?

Distinct keys per window. Low cardinality → permanent-cache candidate. High cardinality → it will never warm.

# *An entity key is a user ID.*
# *So we can't ship it. So we hash it.*

And the hash is what makes it comparable across millions of requests anyway.

| | | |
|---|---|---|
| Entity keys are PII | → | **user IDs, account numbers, tenant IDs.** GDPR and CCPA apply. |
| Can't ship the raw key | → | **hash everything.** |
| The hash | → | **`xxhash` — 64-bit, deterministic, non-reversible.** |
| Deterministic isn't a compromise | → | **same key, same hash, forever, in every process.** |

# One stream, four hashes.

*We learn everything about how your data behaves without ever seeing the data itself.*

## KeyHash

Q · 04 DISTRIBUTION

**Which entity instance.**

`count distinct → key distribution.` 10 keys, a permanent-cache candidate. 5 million, a write-through.

## field_hash

COMPOSITION

**Which leaf field, from which source.**

The route every byte took to land on the response. Cache hit, cache miss, origin — by field.

## OperationHash

Q · 03 ACCESS

**Which operation.**

The persisted-query identity. Which client paths are pulling on this entity, and how often.

## HeaderHash

P · 01 FRAGMENTATION

**Which header variant.**

`count distinct → cache fragmentation.` One bucket per user, or one bucket for everyone.

THE INSTRUMENT

One denormalized event stream · one row per cache op · one discriminator column · 7-day retention · landing in ClickHouse.

`count()` · `uniqExact()`

# Cosmo Entity Intelligence — live.

demo · ~5 min

The four questions become an on-screen checklist.

DEMO FLOW

studio.cosmo · /federation/employees

**01** **Open the federation graph heatmap.**
Switch metrics: requests · latency · cache hits.

✓ **Accessed?**
Requests heatmap — `~40,000 reads/day`.
Q · 03

✓ **Key distribution?**
`uniqExact(KeyHash) → 10 keys` — permanent-cache candidate.
Q · 04

✓ **Updated?**
Mutation events — `~6% writes / hour`.
Q · 02

✓ **Cacheable?**
The verdict — open the `Employee` entity detail panel.
Q · 01

THE SETUP

- Cosmo's public `employees` demo graph.
- One entity, `Employee` keyed `@key(fields: "id")`, across **6 subgraphs**.
- `k6` traffic, **~92% reads / 8% writes**.
- Hot keyspace of **10 IDs**.

# *A dashboard describes.*
# *This one prescribes.*

The loop closes — telemetry becomes a schema change.

THE VERDICT — ONE SENTENCE PER ENTITY

● STRONG CANDIDATE

**`Employee`** · Strong candidate. `40,000` reads/day, `6%` writes, `0%` errors. Textbook caching candidate.

*Every word traces back to a* `count()`.

RECOMMENDATIONS FEED · RANKED

Apply all — rewrites SDL →

**01** `low_cardinality`
≤ 50 distinct keys over 1000+ reads — permanent-cache candidate.
closes P · 04

**02** `invalidation_gap`
12 of 18 mutations carry `@cacheInvalidate` — unwired writes leak stale reads.
closes P · 02

# *Federation gives you a vocabulary.*
# *The router gives you hashes.*
# *The story is the join.*

Federation's vocabulary is tiny — entities and keys.

We used it for resolving entities. Then for streams. Then for caching.

And now that same key, **hashed**, is a measurement instrument.

Four questions. One event stream. A couple of hashes.

## Understanding your graph — one hash at a time.

**Jens Neuse**
WunderGraph

# Questions?

org · WunderGraph · product · Cosmo Entity Intelligence · docs · cosmo-docs.wundergraph.com