# Easy Freshness with Pequod Cache Joins

Bryan Kate, Eddie Kohler, Mike Kester
Harvard University

Yandong Mao, Neha Narula, Robert Morris
MIT

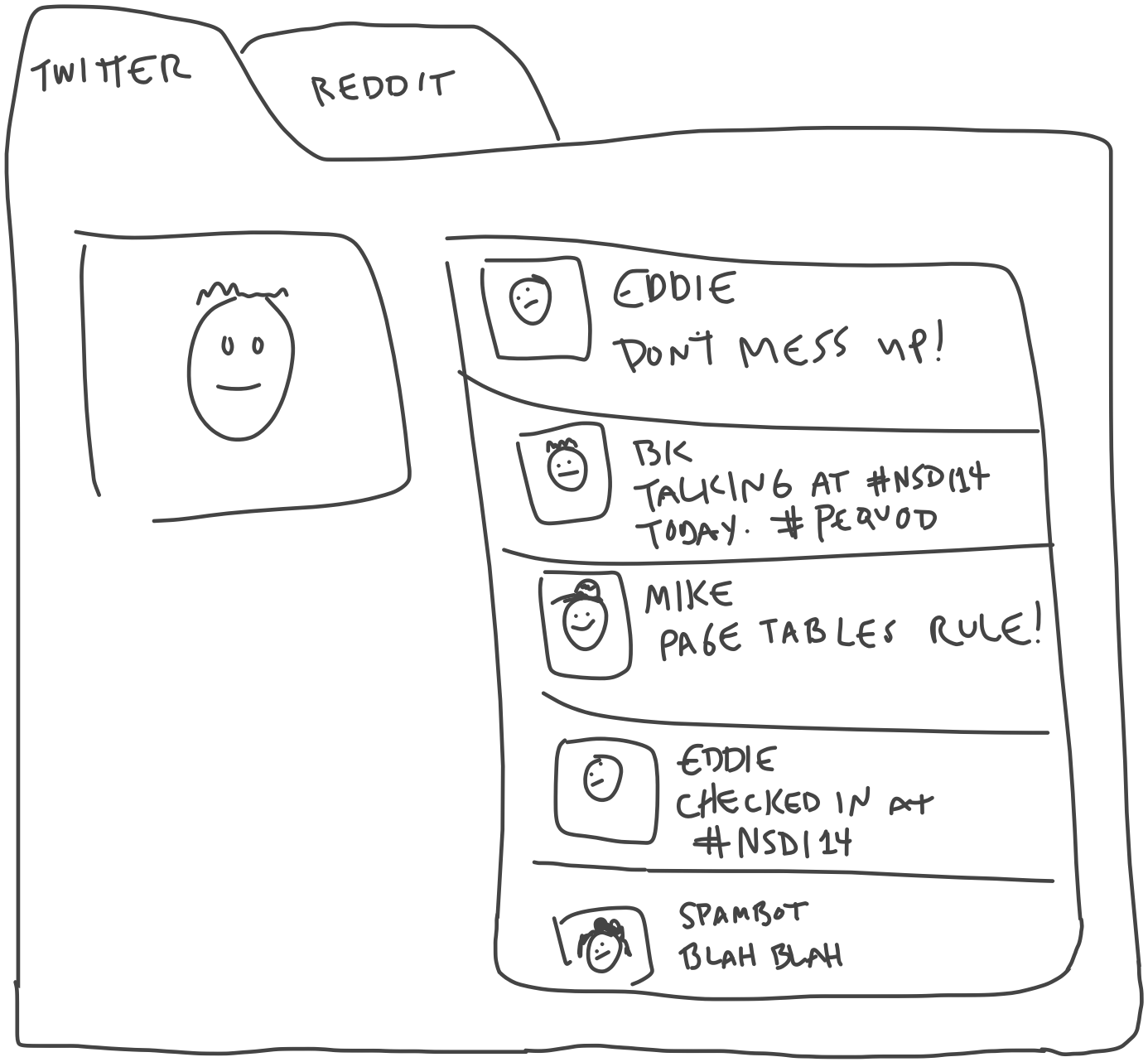
# tl;dr

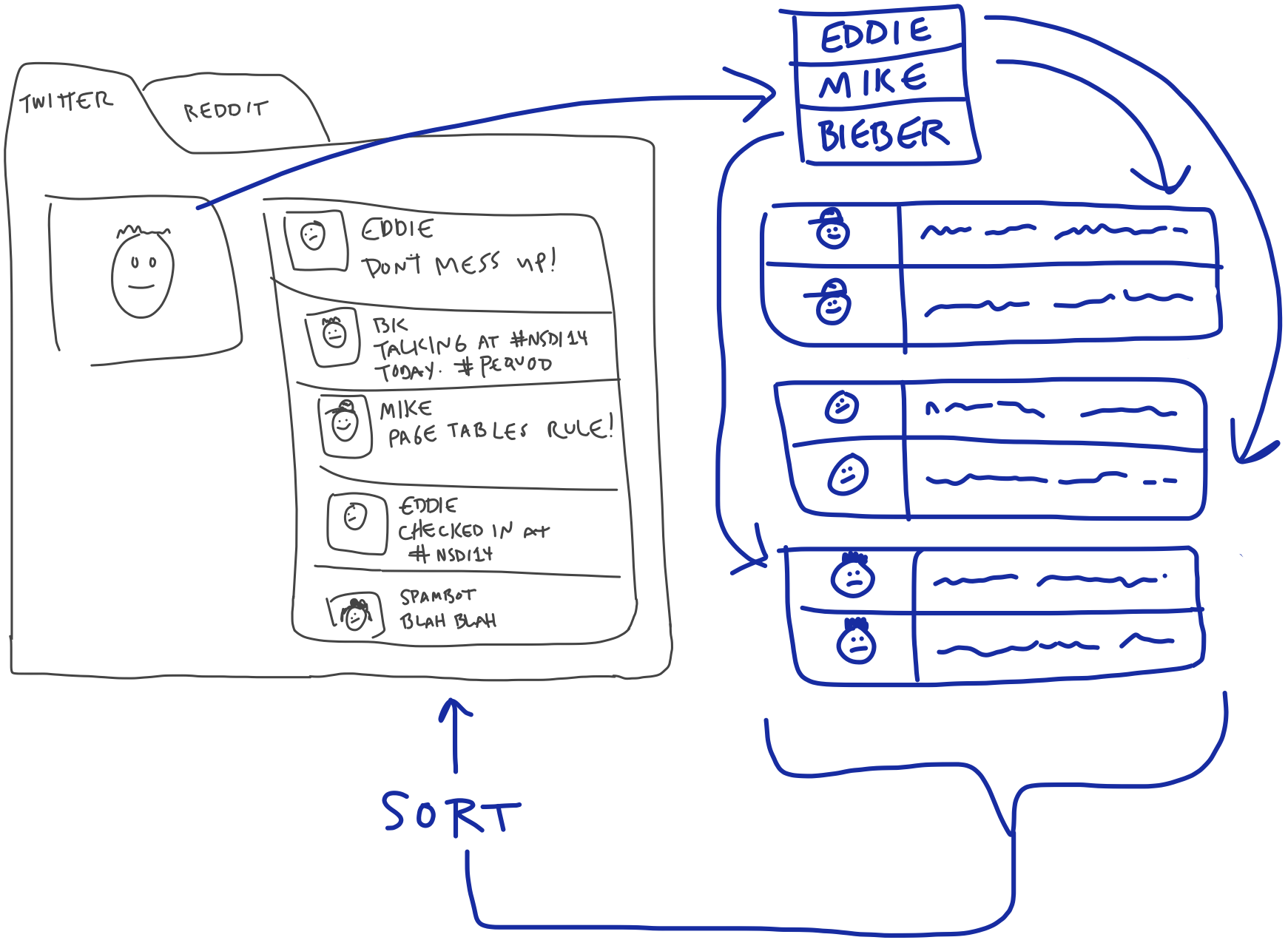
Web application caches should support materialized views natively.

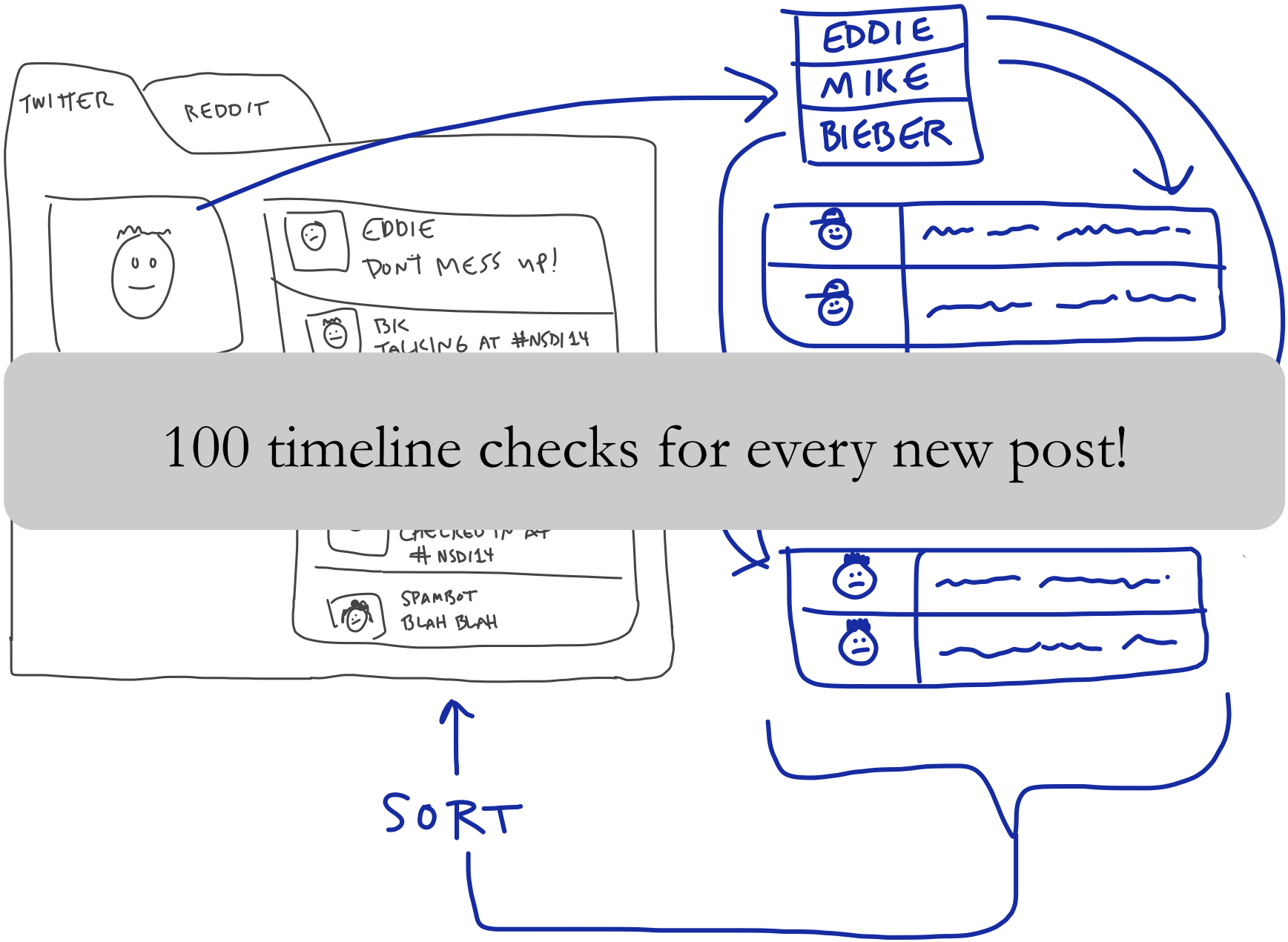
In-cache materialized views are easy to use and have good performance.

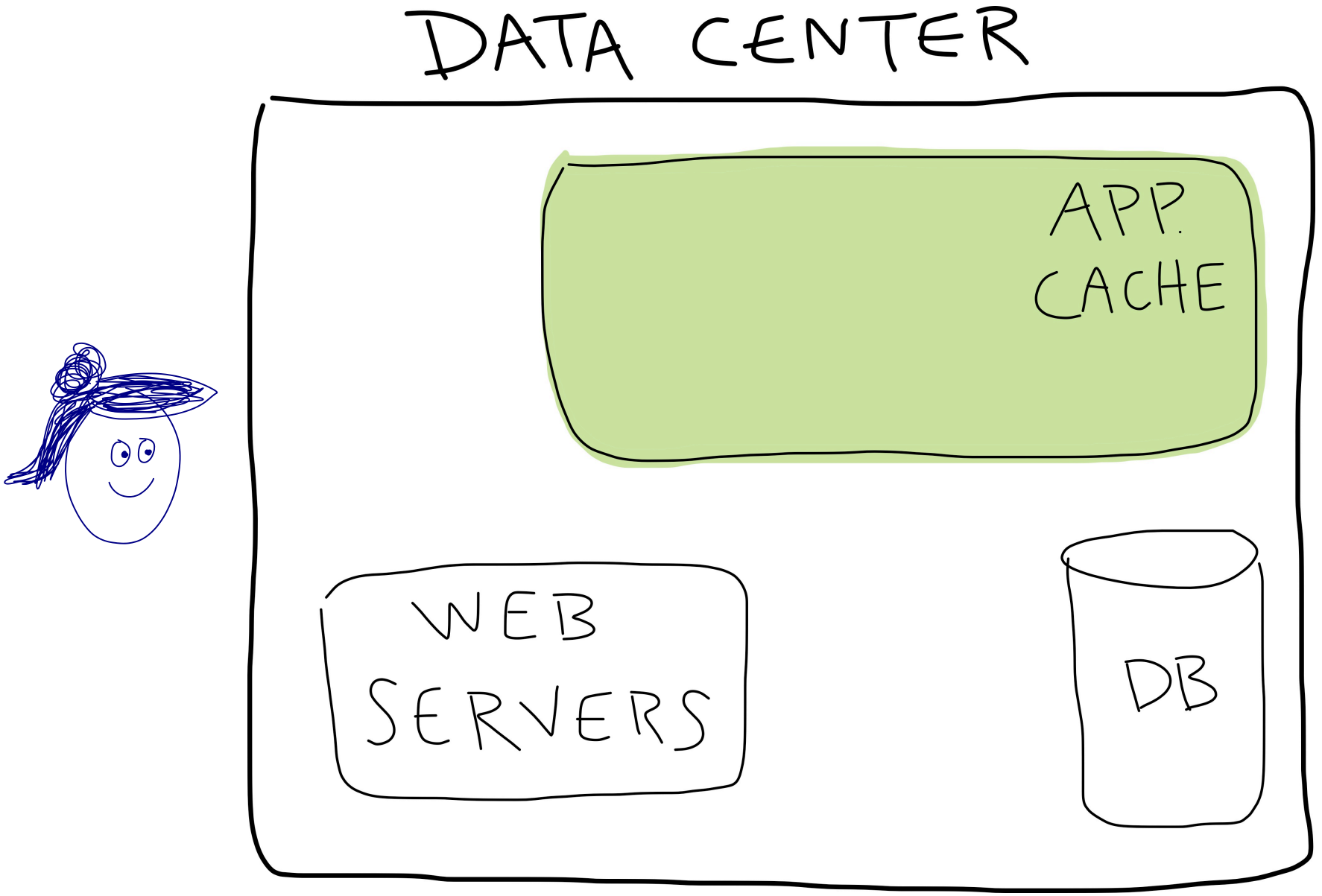
# application cache

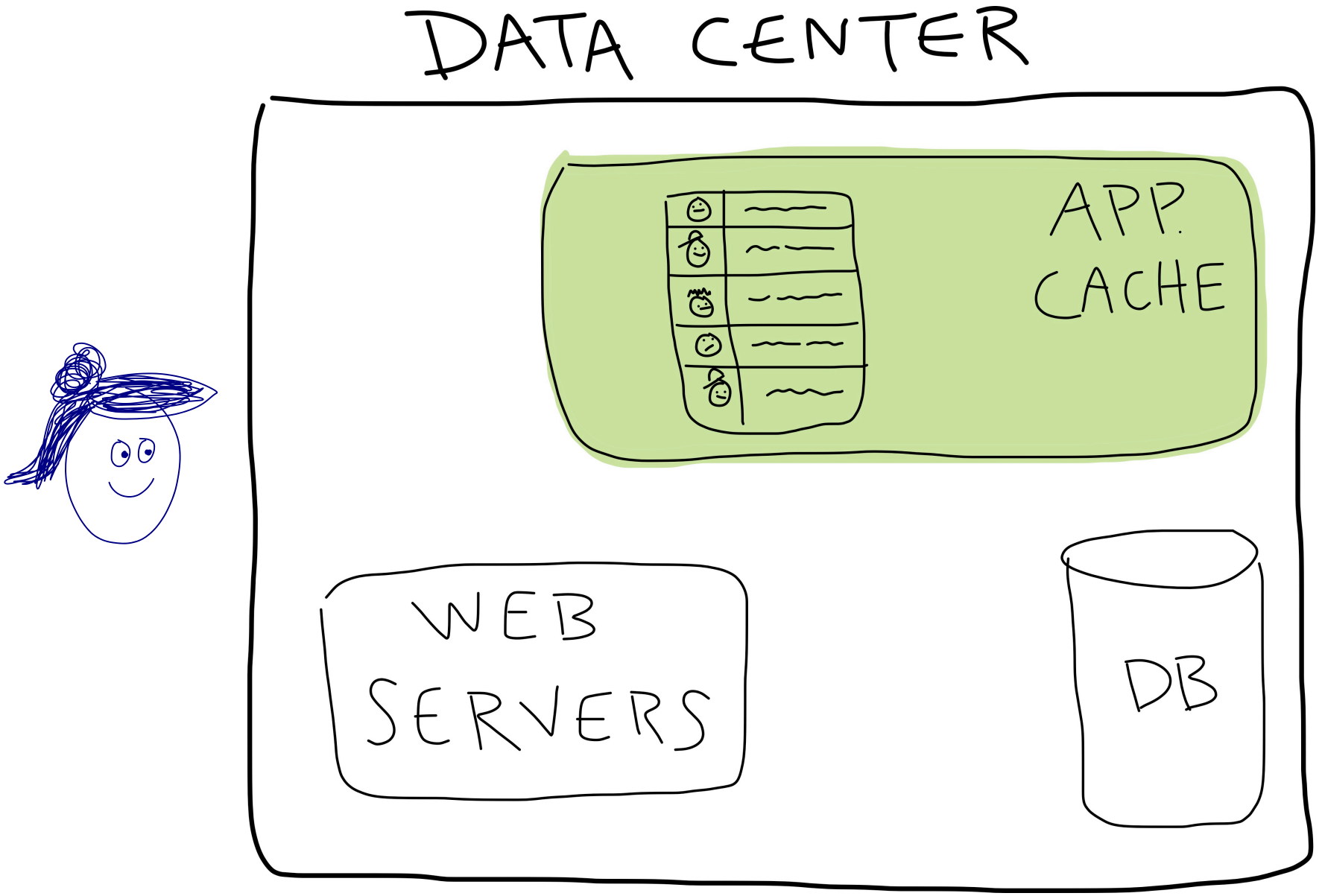
- fast key-value cache
  - examples: memcached, Redis
- offloads reads from database
- managed by application developer
  - assume burden of maintenance

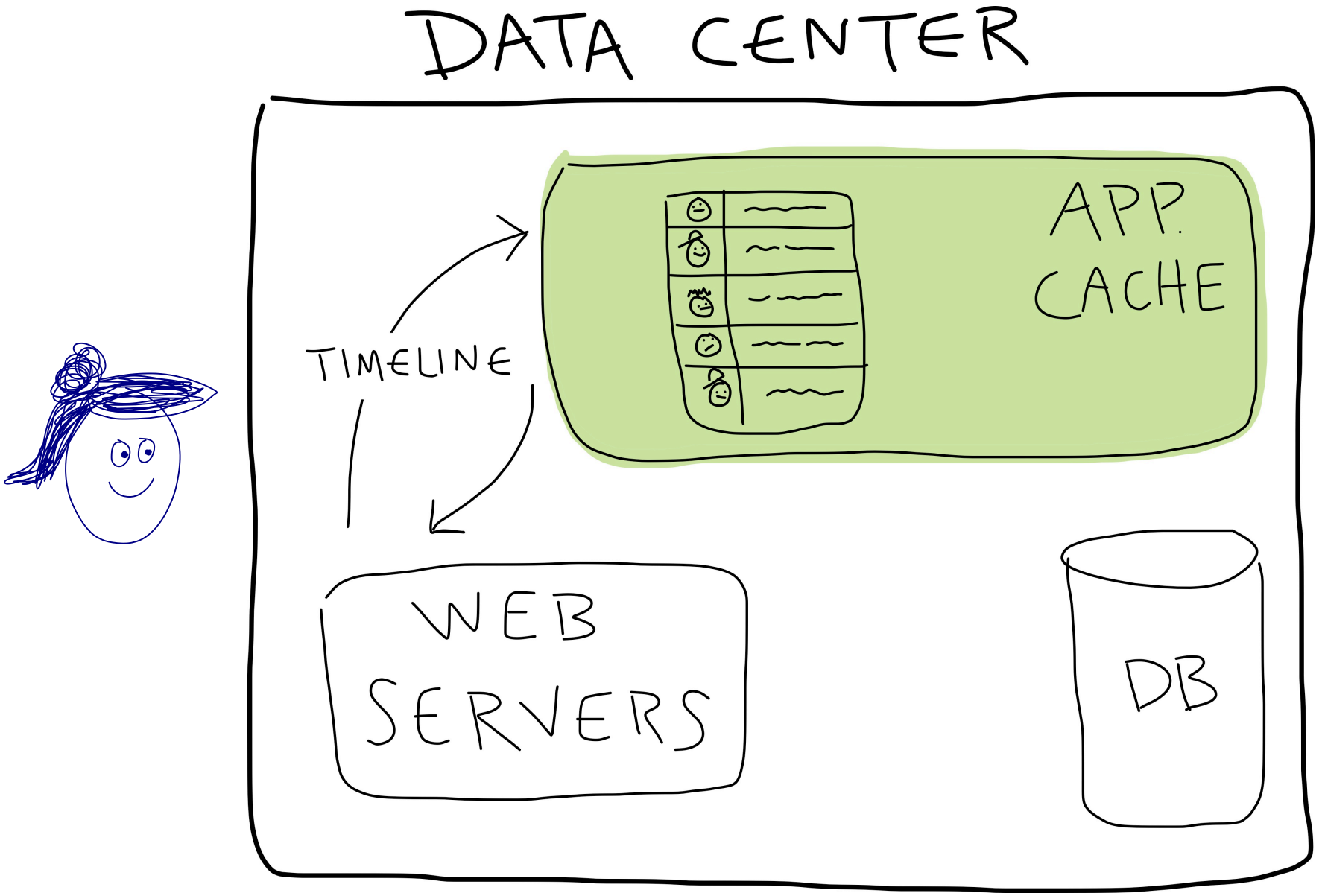

TWITTER
REDDIT
EDDIE
DON'T MESS UP!
BK
TALKING AT #NSDI14
TODAY. #PEQUOD
MIKE
PAGE TABLES RULE!
EDDIE
CHECKED IN AT
#NSDI14
SPAMBOT
BLAH BLAH

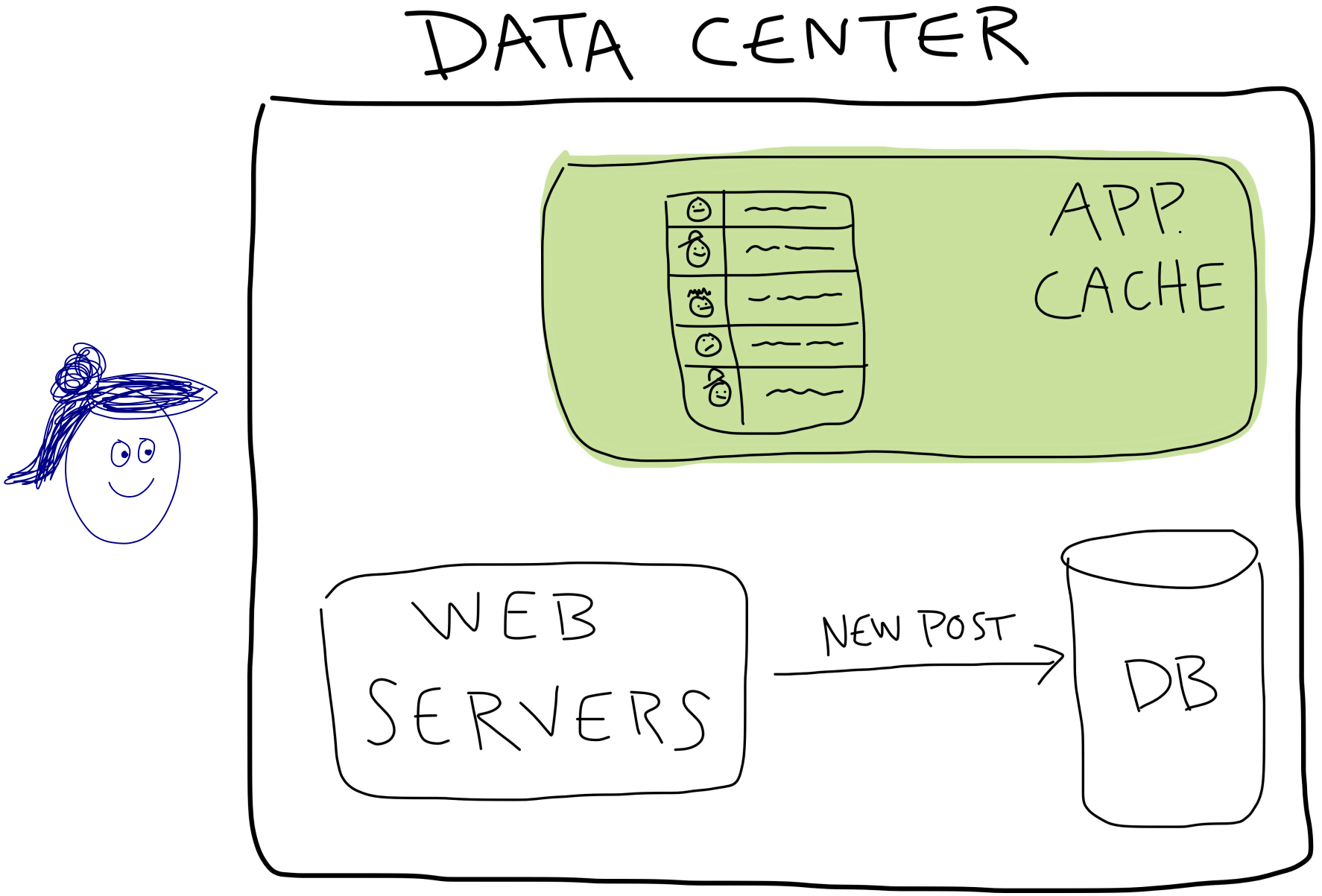

TWITTER
REDDIT
EDDIE
DON'T MESS UP!
BIK
TALKING AT #NSDI14
TODAY. #PERIOD
MIKE
PAGE TABLES RULE!
EDDIE
CHECKED IN AT
#NSDI14
SPAMBOT
BLAH BLAH
EDDIE
MIKE
BIEBER
SORT

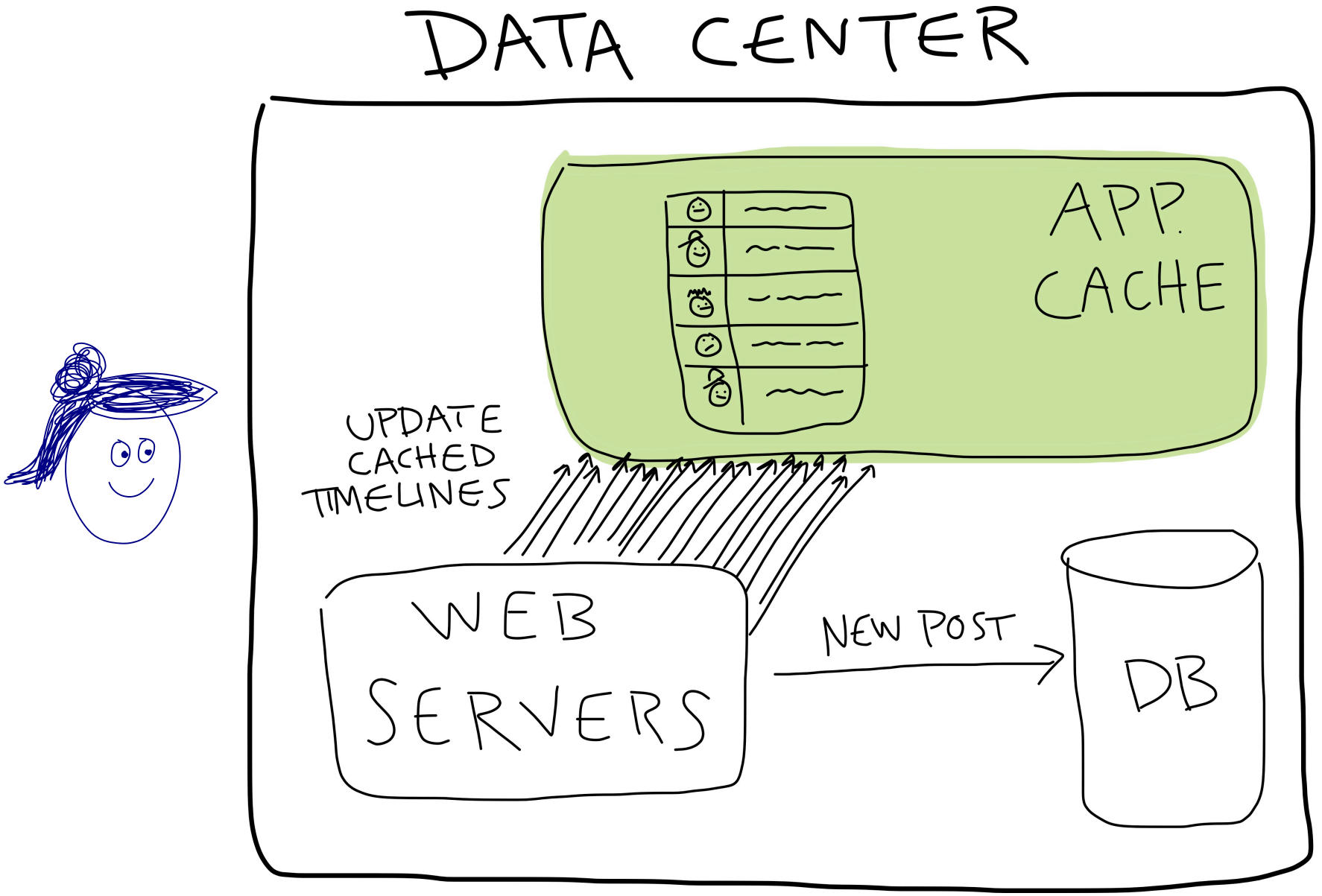

TWITTER
REDDIT
EDDIE
DON'T MESS UP!
BIK
#NSDI14
SPAMBOT
BLAH BLAH
EDDIE
MIKE
BIEBER
100 timeline checks for every new post!
SORT

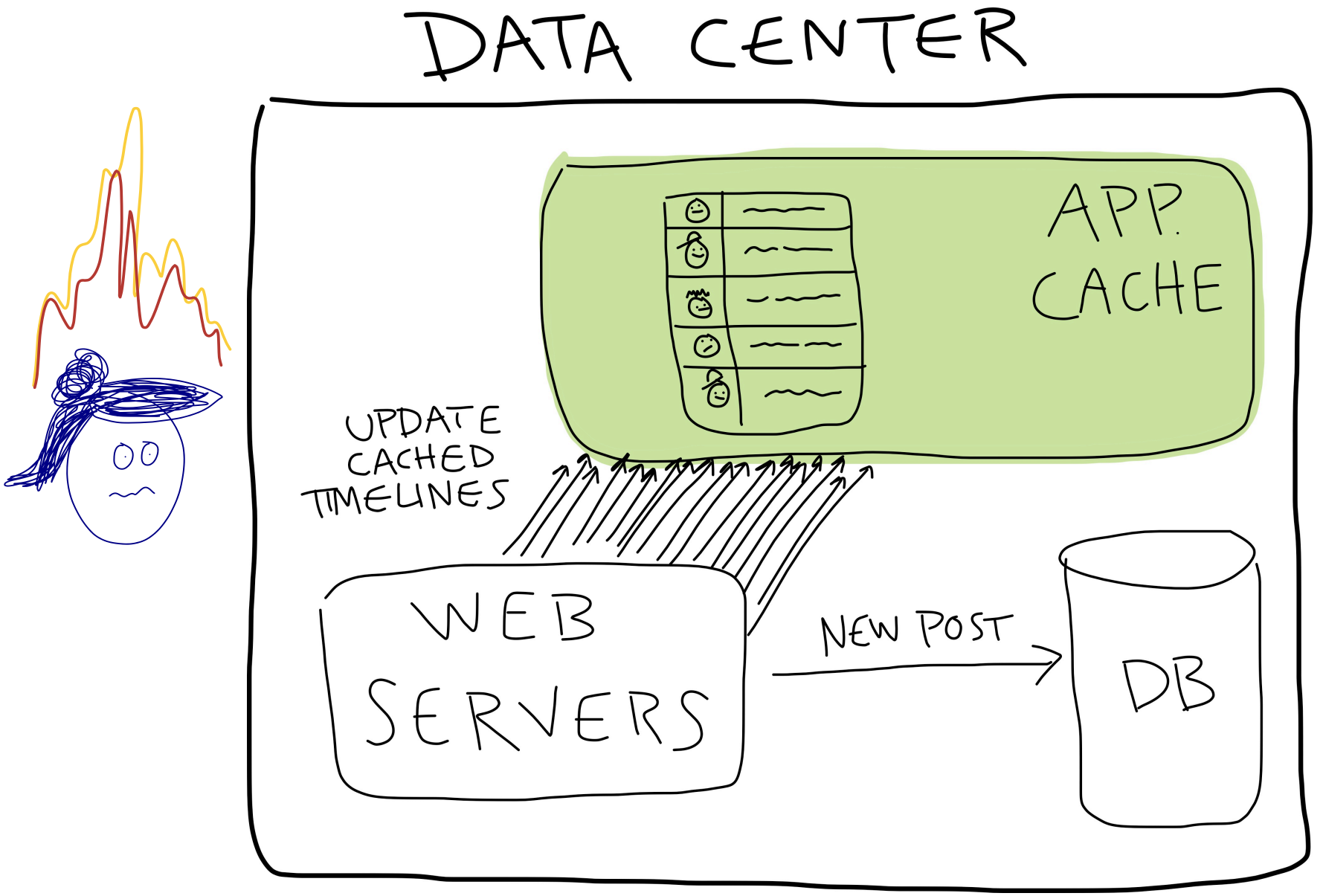

DATA CENTER
APP.
CACHE
WEB
SERVERS
DB


DATA CENTER
APP.
CACHE
WEB
SERVERS
DB


DATA CENTER
APP.
CACHE
TIMELINE
WEB
SERVERS
DB


DATA CENTER
APP.
CACHE
WEB
SERVERS
NEW POST
DB


DATA CENTER
APP. CACHE
UPDATE CACHED TIMELINES
WEB SERVERS
NEW POST
DB


DATA CENTER
APP.
CACHE
UPDATE
CACHED
TIMELINES
WEB
SERVERS
NEW POST
DB

# timeline database query

```
SELECT post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster
      AND sub.user = 'bk'
      AND post.time >= 100
  ORDER BY post.time;
```


DB

# timeline materialized view

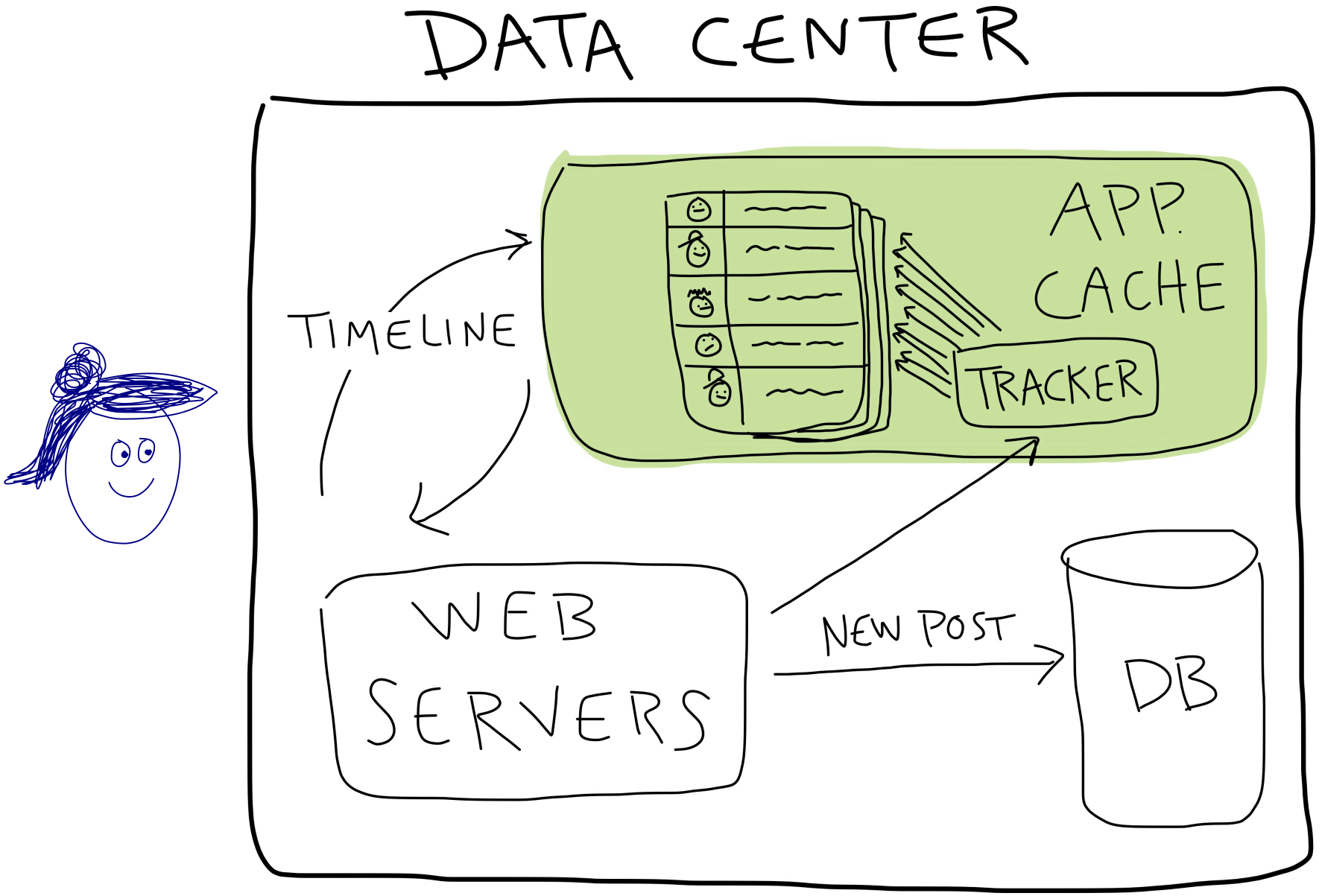
```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user, post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster;

SELECT * FROM tline
  WHERE tline.user = ‘bk’ AND tline.time >= 100
  ORDER BY tline.time;
```

- arrange data for quick reading
  - computation happens in advance—good!
  - simple query on materialized data—good!


DB

# easy, but slow

- the database becomes a bottleneck
  - most important job: durable storage
  - handling reads + writes may be too much
  - better to offload reads
  - implementation issues (locks, transactions, …)


DATA CENTER
APP.
CACHE
TRACKER
TIMELINE
WEB
SERVERS
NEW POST
DB

# Pequod

- a distributed application cache
- materialized views in a key-value cache
  - operations: get, put, scan, plus **join**
- good performance and programmability

# advanced materialized views

- simple materialized views are a bad fit for caches
  - need advanced features from recent research

- **partial**: only portions are materialized as needed
- **dynamic**: portions are selected based on requests
- **incremental updates**: track dependencies between data
- **eager** updates
- **lazy** updates
- **distributed**
- in an ordered **key-value cache!**

# KV materialized views?

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user. post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster;
```

- but Pequod only understands get, put, scan!
  - want key-value for performance
  - how to represent the relations needed for views?

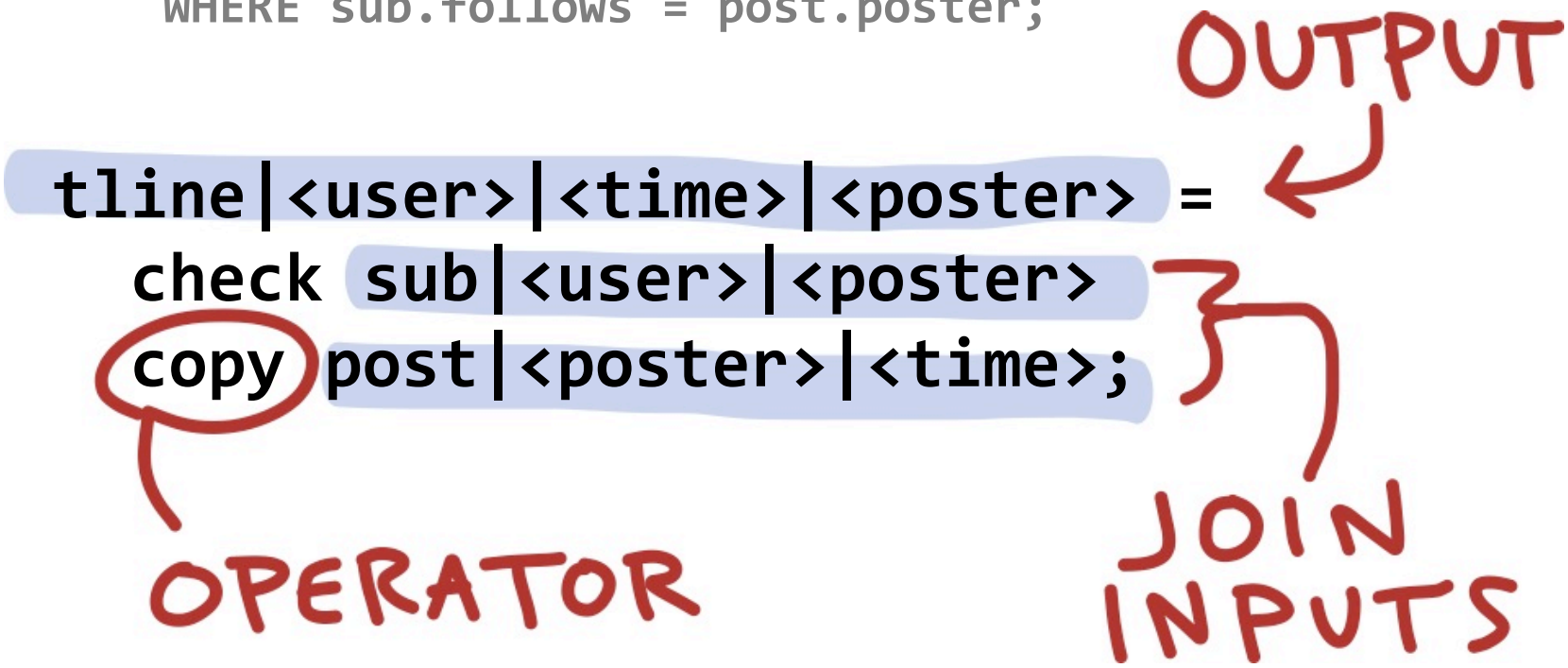
# Pequod cache joins

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user. post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster;
```

```
tline|<user>|<time>|<poster> =
  check sub|<user>|<poster>
  copy post|<poster>|<time>;
```

# Pequod cache joins

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user. post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster;
```

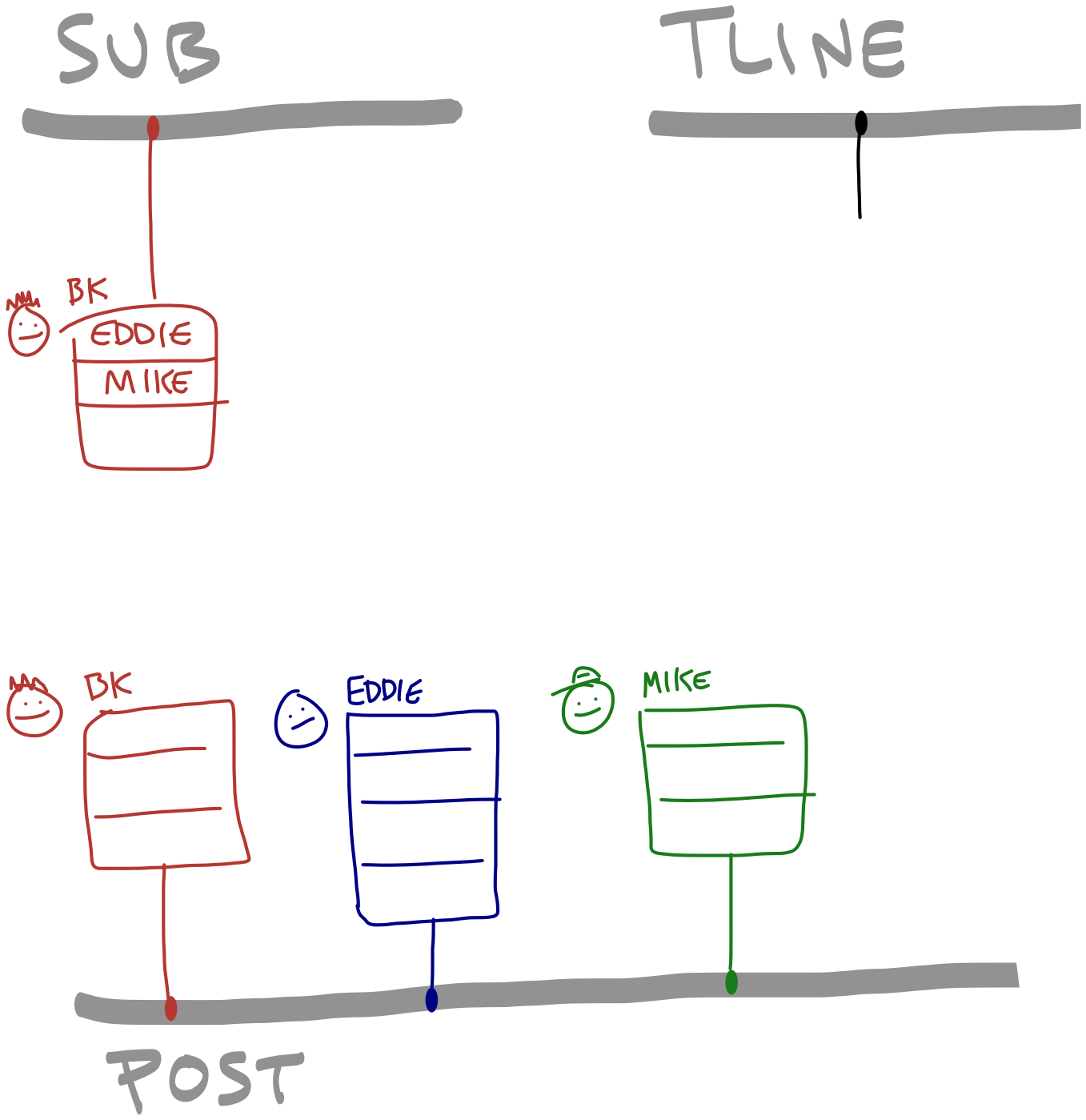


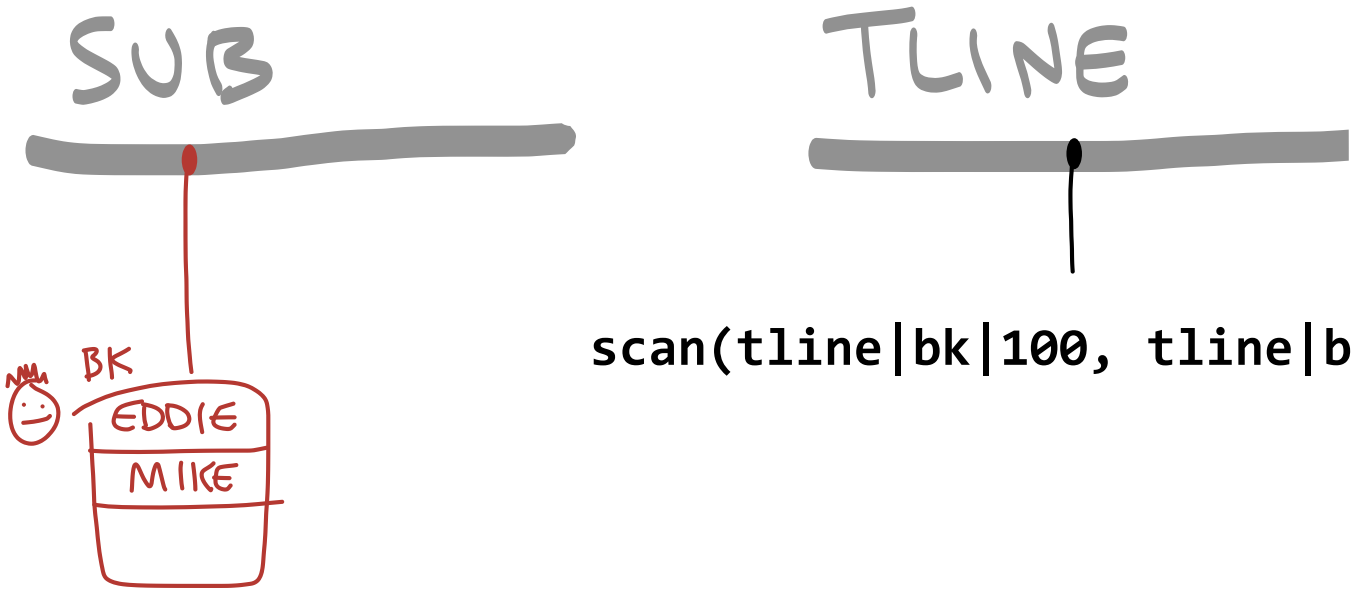
# Pequod cache joins

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user. post.time, post.poster, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.poster;
```

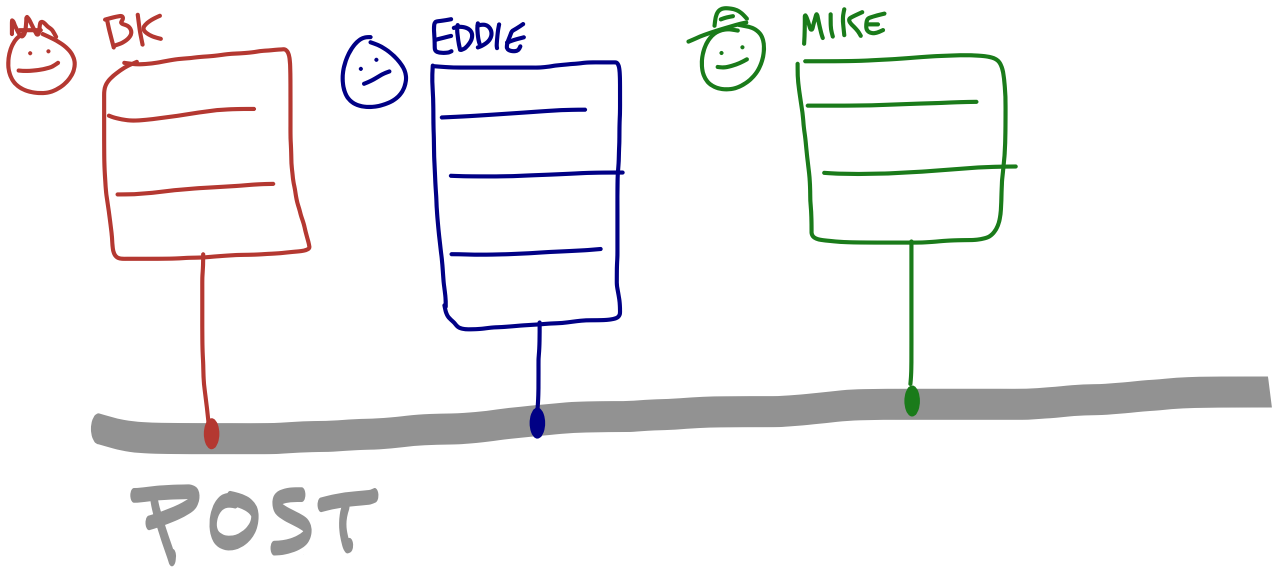
```
tline|<user>|<time>|<poster> =
  check sub|<user>|<poster>
  copy post|<poster>|<time>;
```

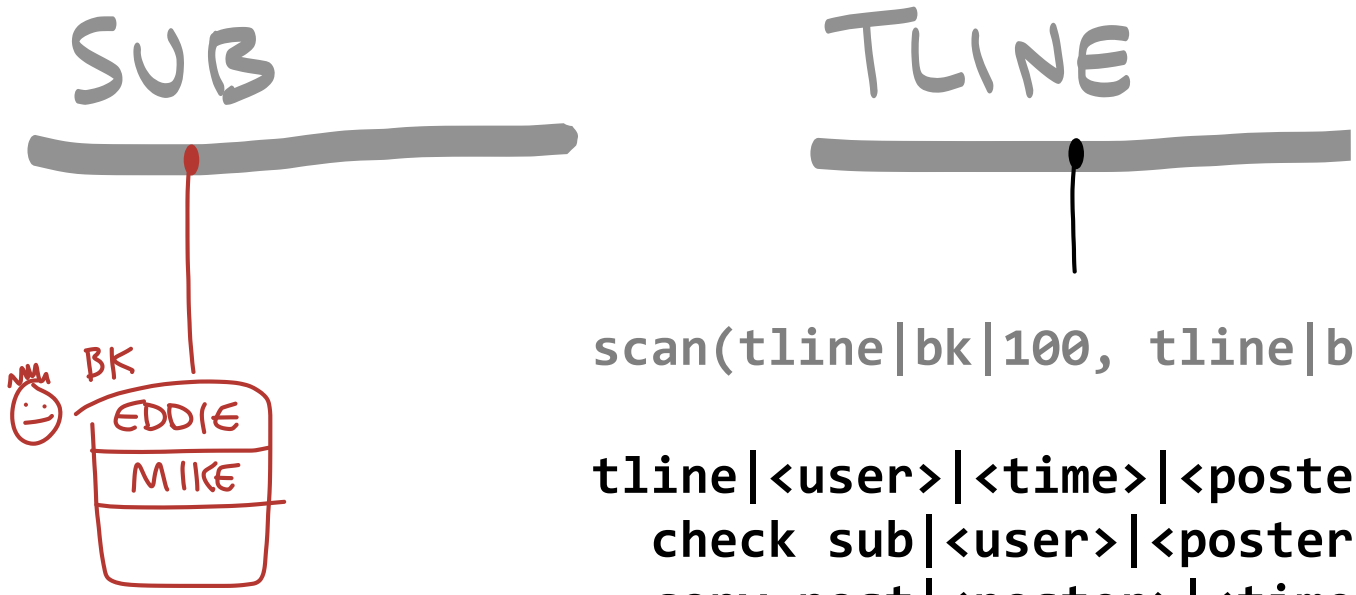
```
scan(tline|bk|100, tline|bk∞)
```


SUB
TLINE
BK
EDDIE
MIKE
BK
EDDIE
MIKE
POST

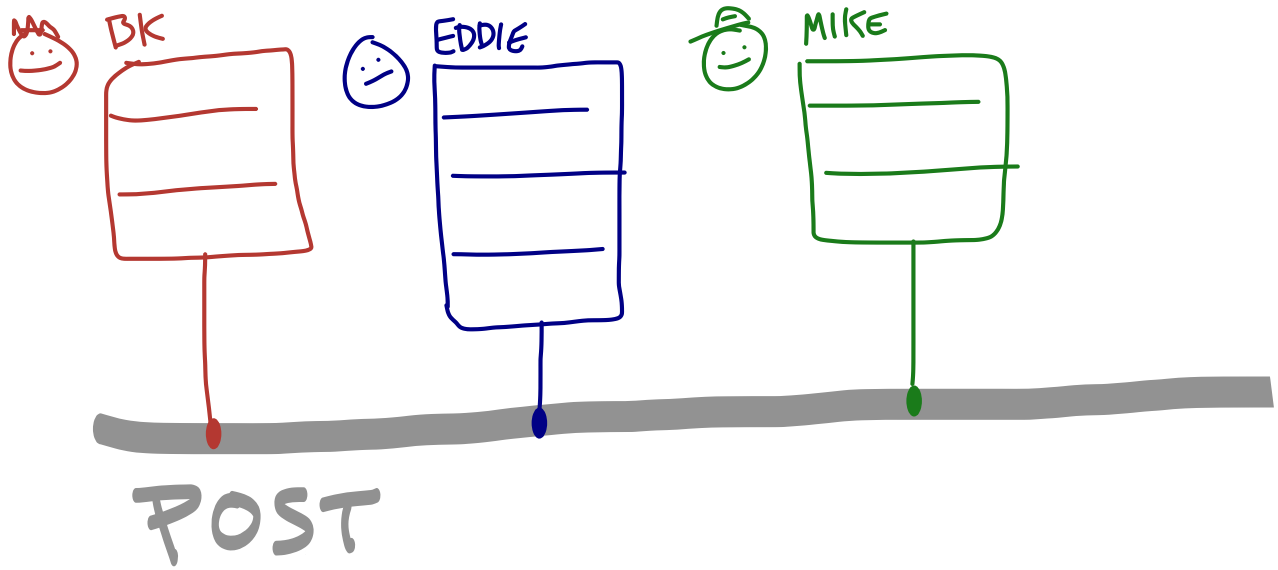


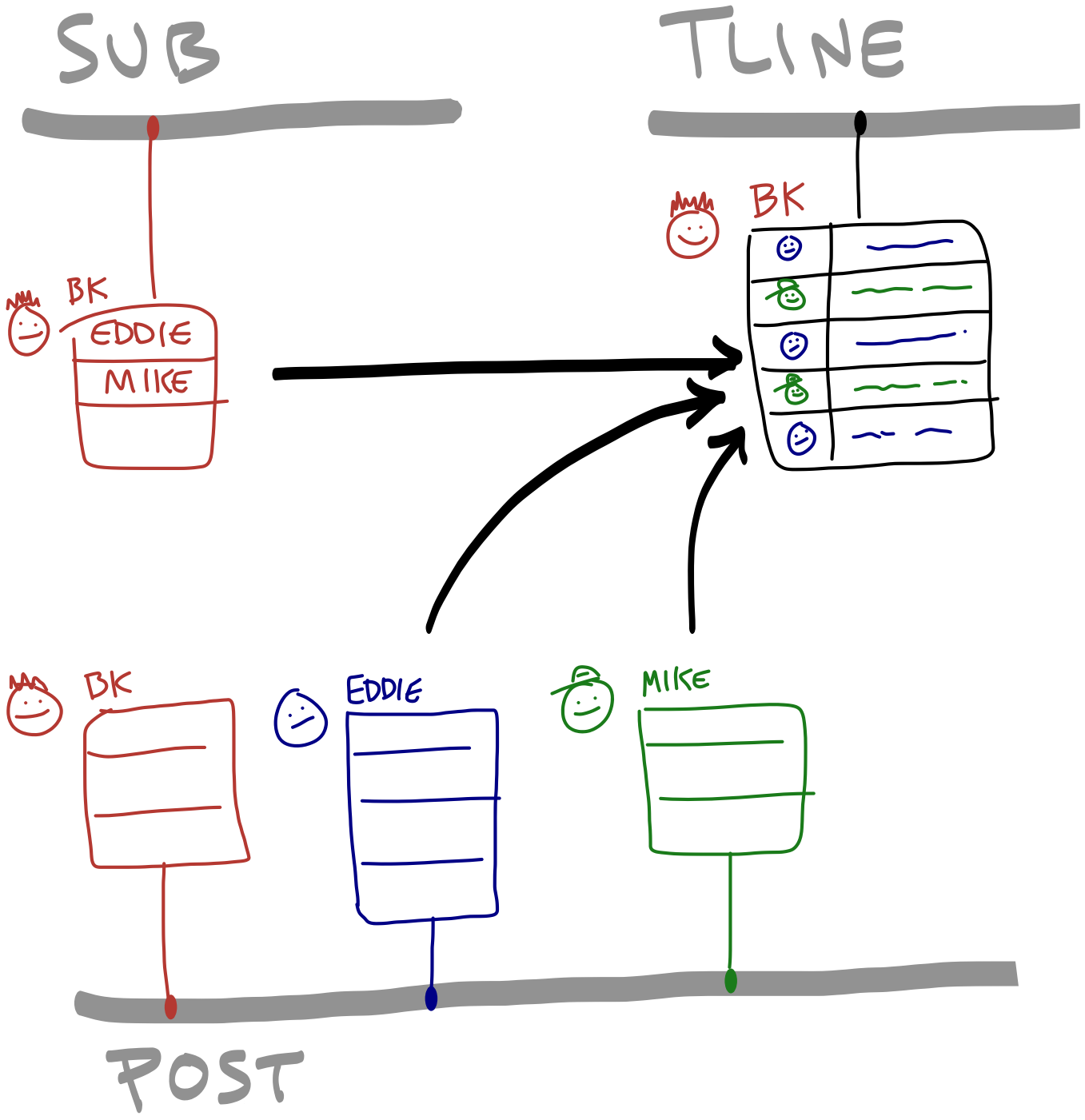

```
scan(tline|bk|100, tline|bk∞)
```

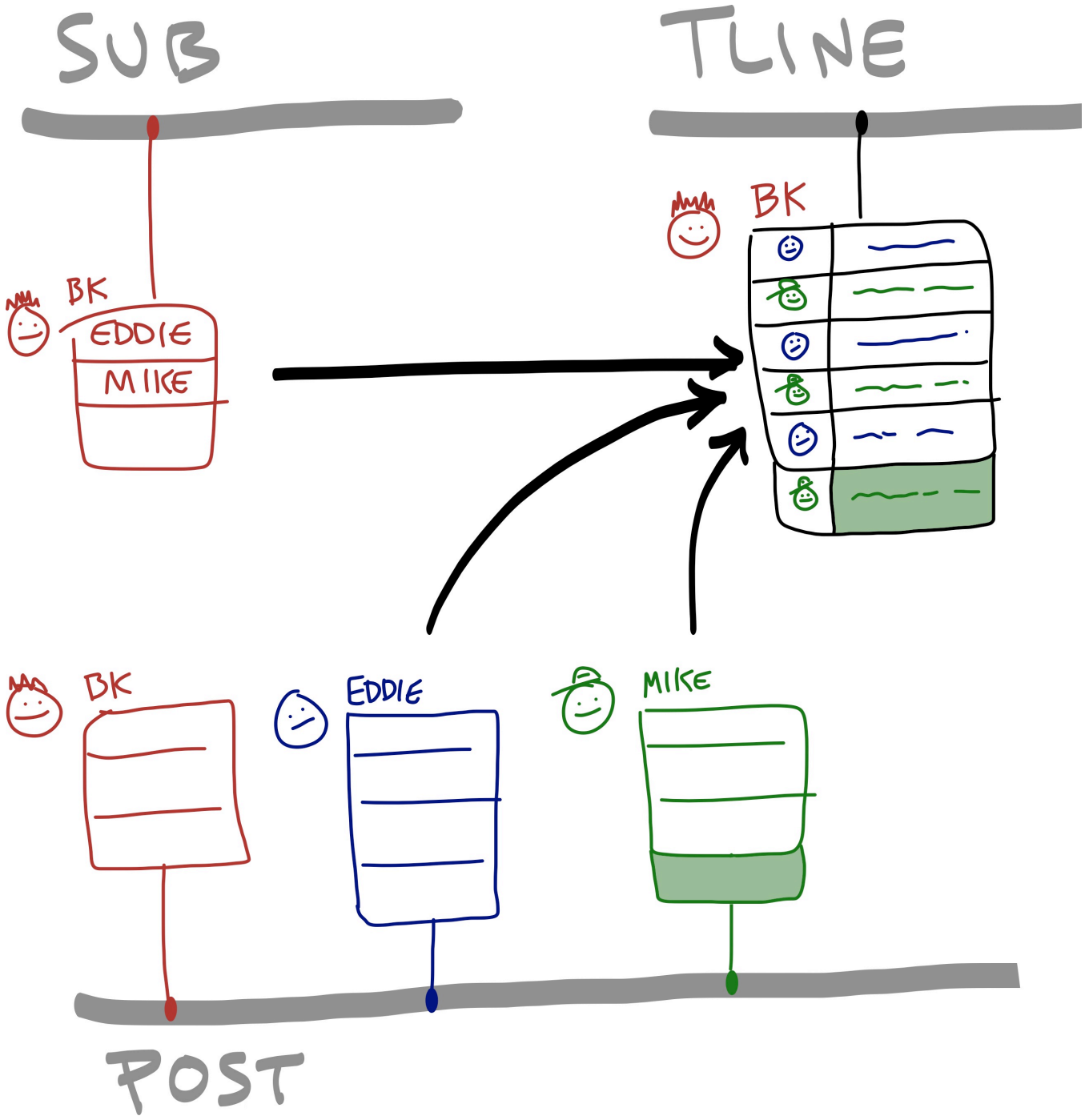


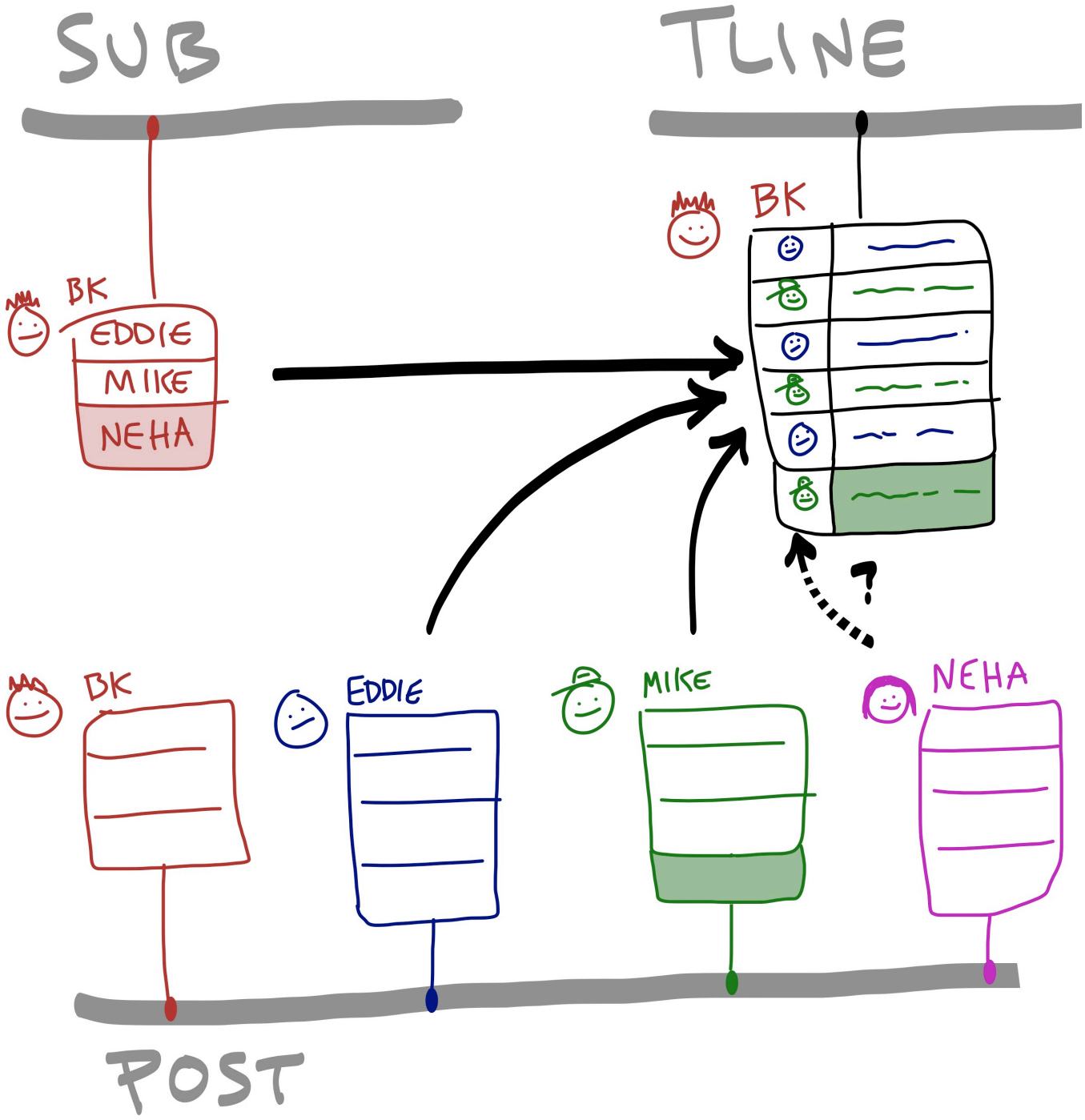



```
scan(tline|bk|100, tline|bk∞)

tline|<user>|<time>|<poster> =
  check sub|<user>|<poster>
  copy post|<poster>|<time>;
```




SUB
TLINE
BK
EDDIE
MIKE
BK
BK
EDDIE
MIKE
POST

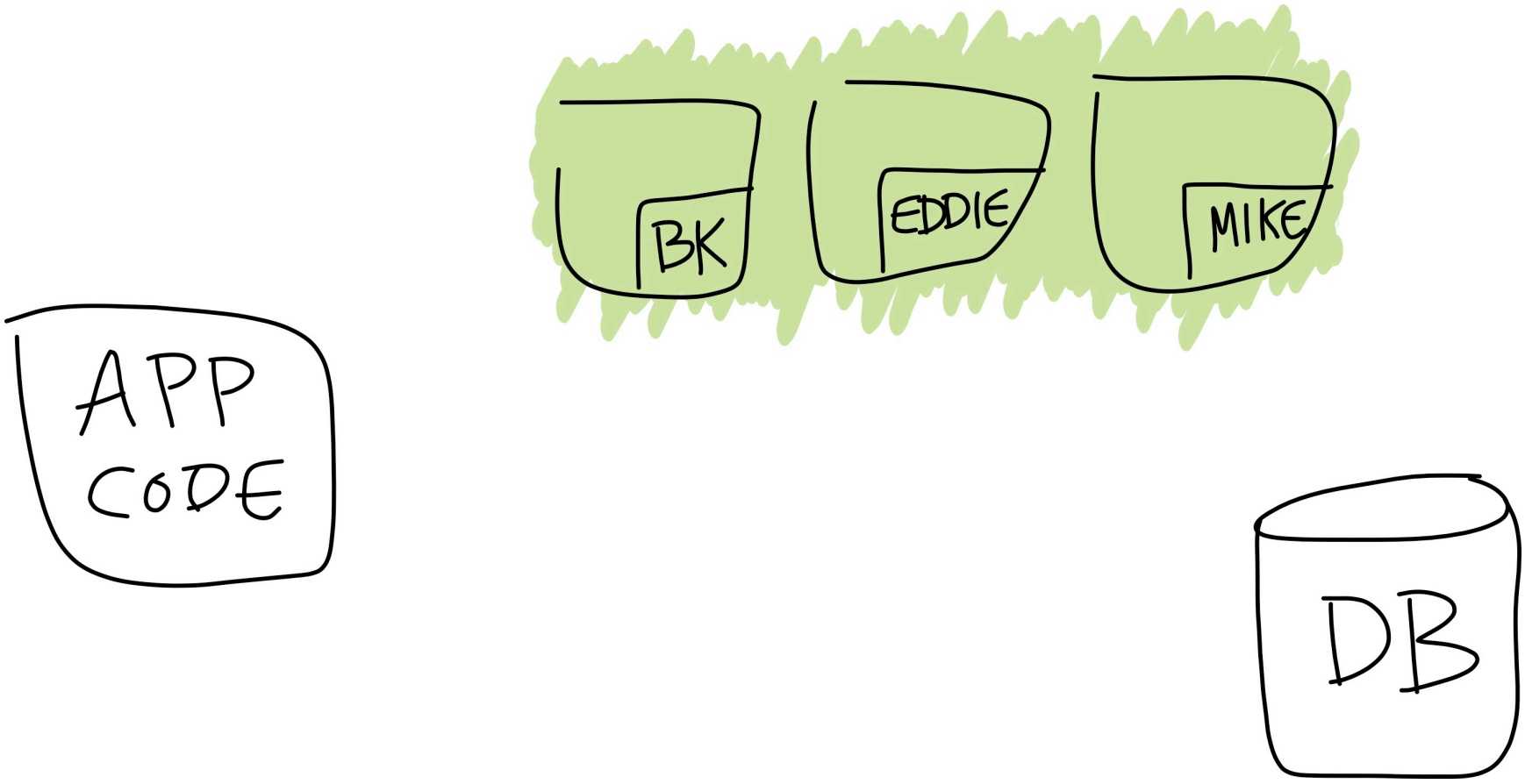

SUB
TLINE
BK
EDDIE
MIKE
BK
BK
EDDIE
MIKE
POST


SUB
TLINE
BK
EDDIE
MIKE
NEHA
BK
?
BK
EDDIE
MIKE
NEHA
POST

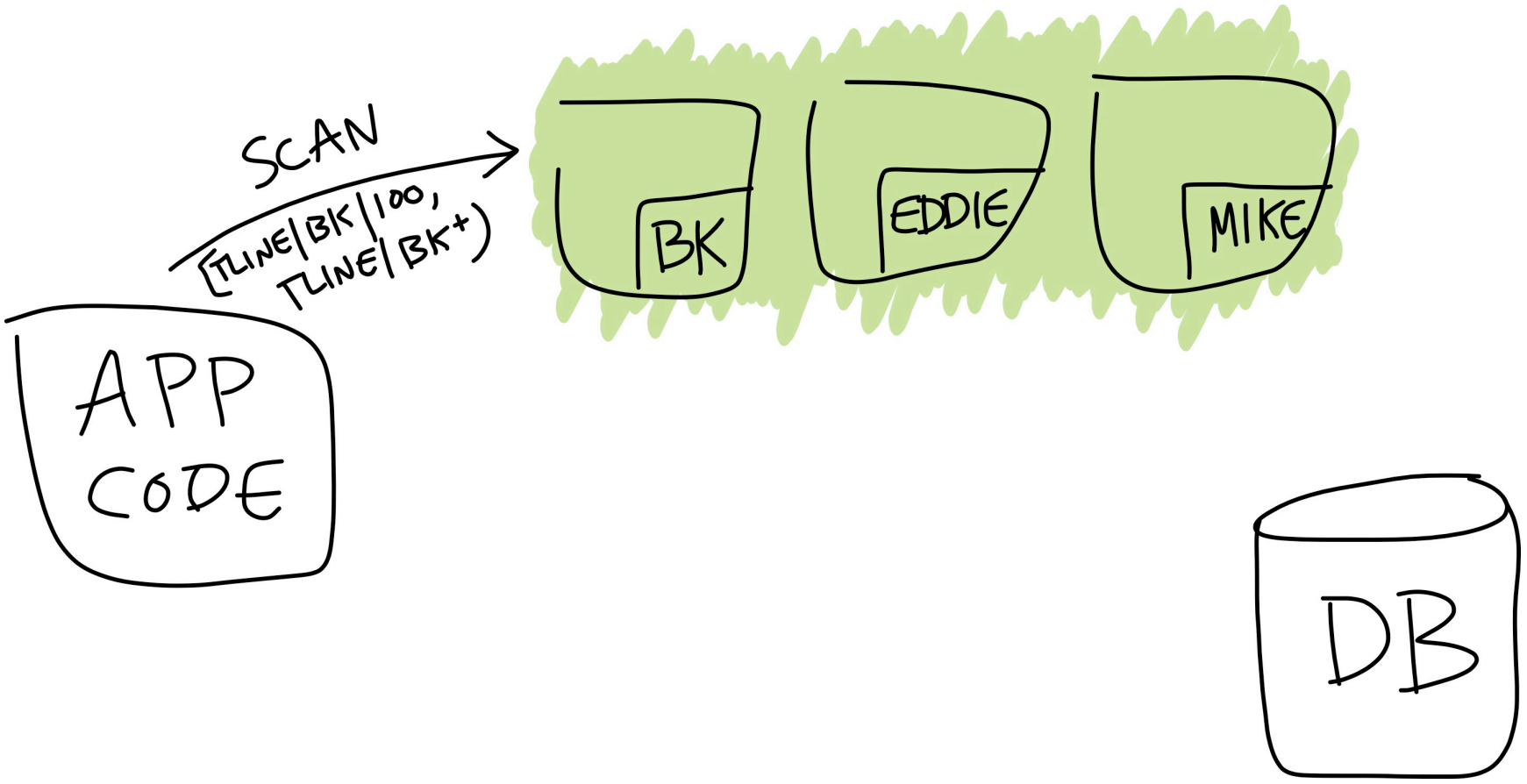
# scale

- distributed Pequod scales to large data sets
  - key design choice: computation is local
- base data is partitioned
  - example: sub, post "tables"
- cache joins can be computed anywhere
  - base data transparently replicated as necessary

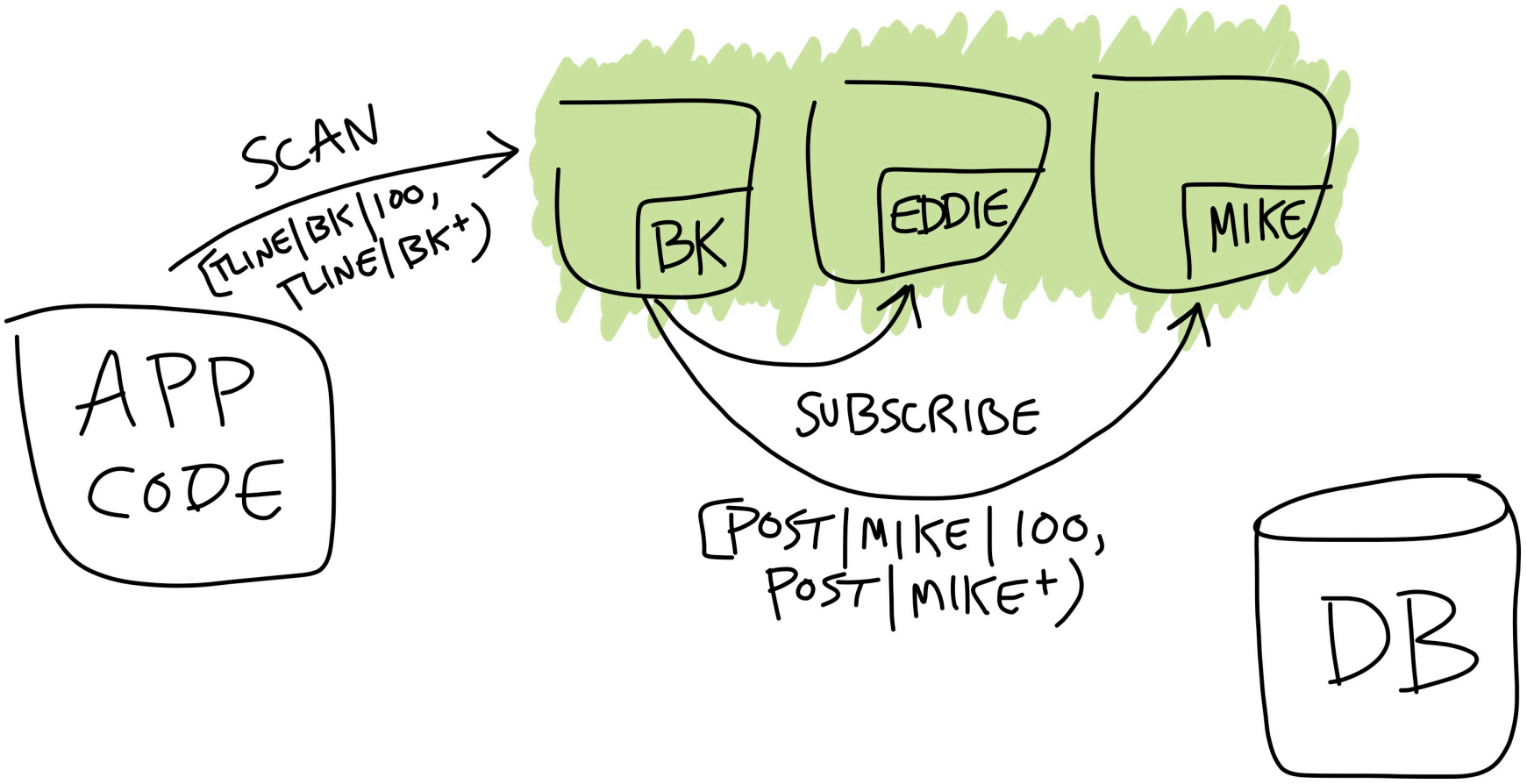
# distributed deployment



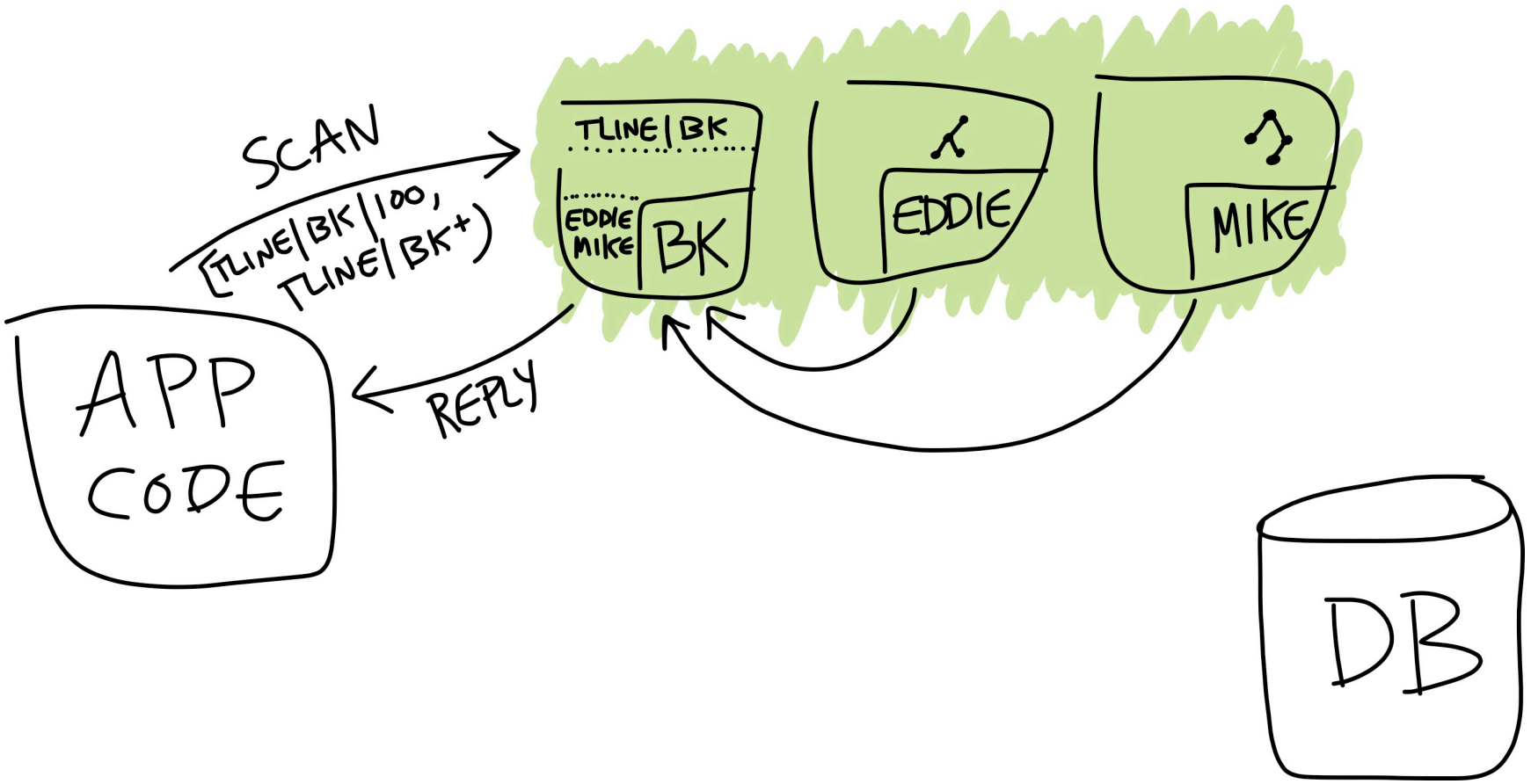
# distributed deployment (read)



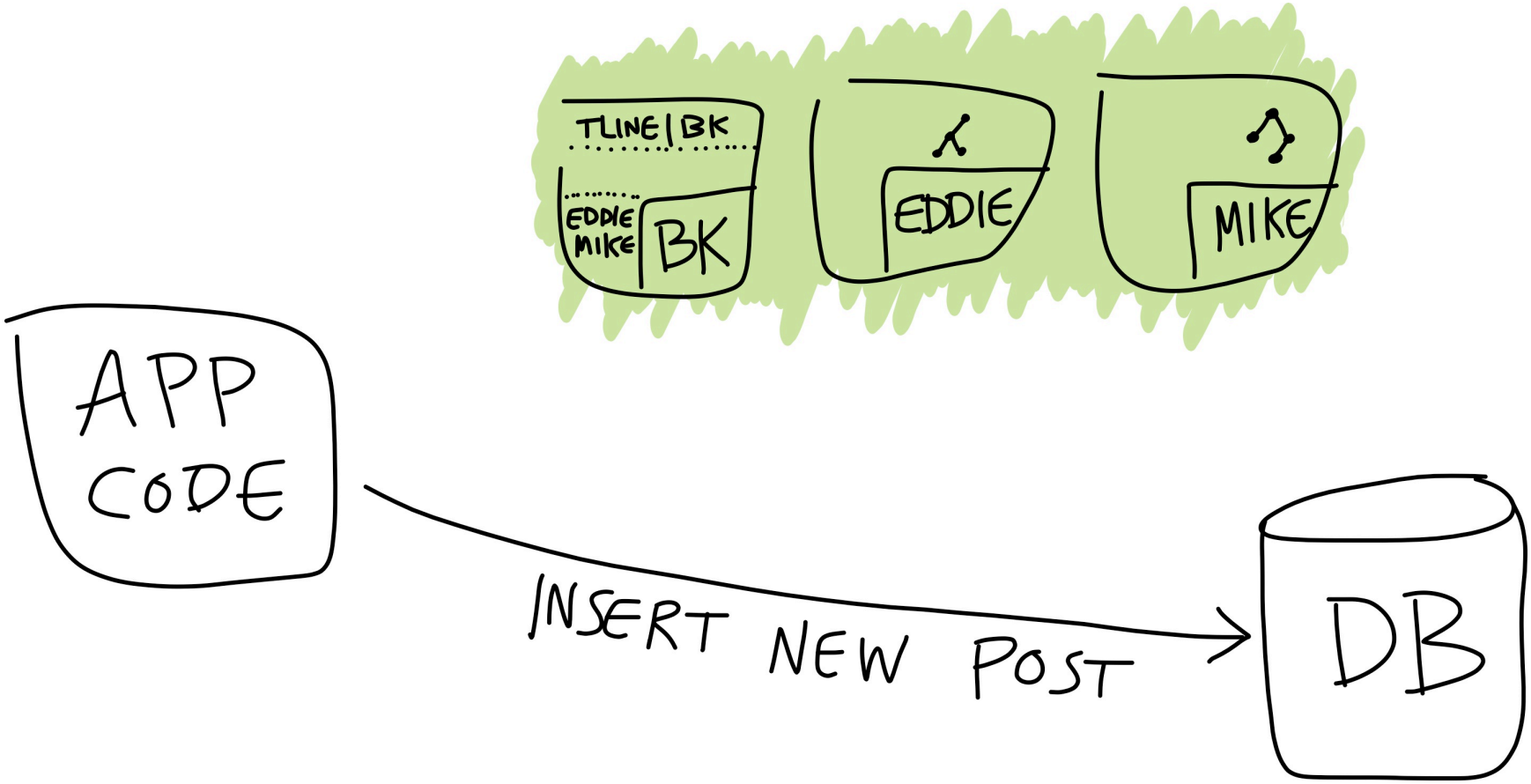
# distributed deployment (read)



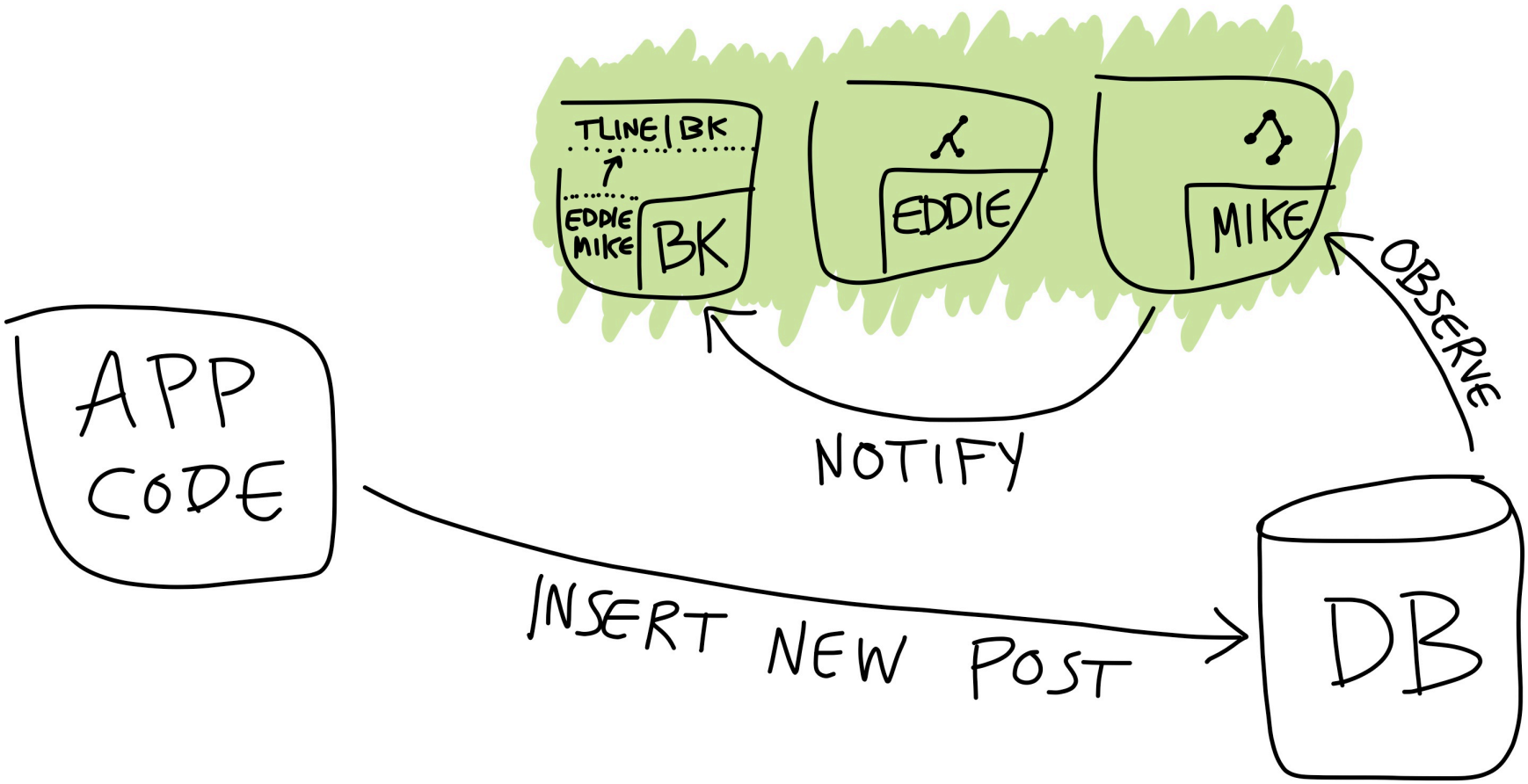
# distributed deployment (read)



# distributed deployment (write)



# distributed deployment (write)



# other features

- advanced cache joins
  - interleaved: collocate different kinds of data
  - stacked
  - materialized, non-materialized, or snapshot
  - aggregates
- eviction
- consistency

# evaluation

- Twitter-like benchmark
  - based on 2009 Twitter social graph
  - check, subscribe, post (100:10:1)

- evaluate potential bottlenecks in Pequod
  - database omitted in experiments
  - clients write data directly to Pequod

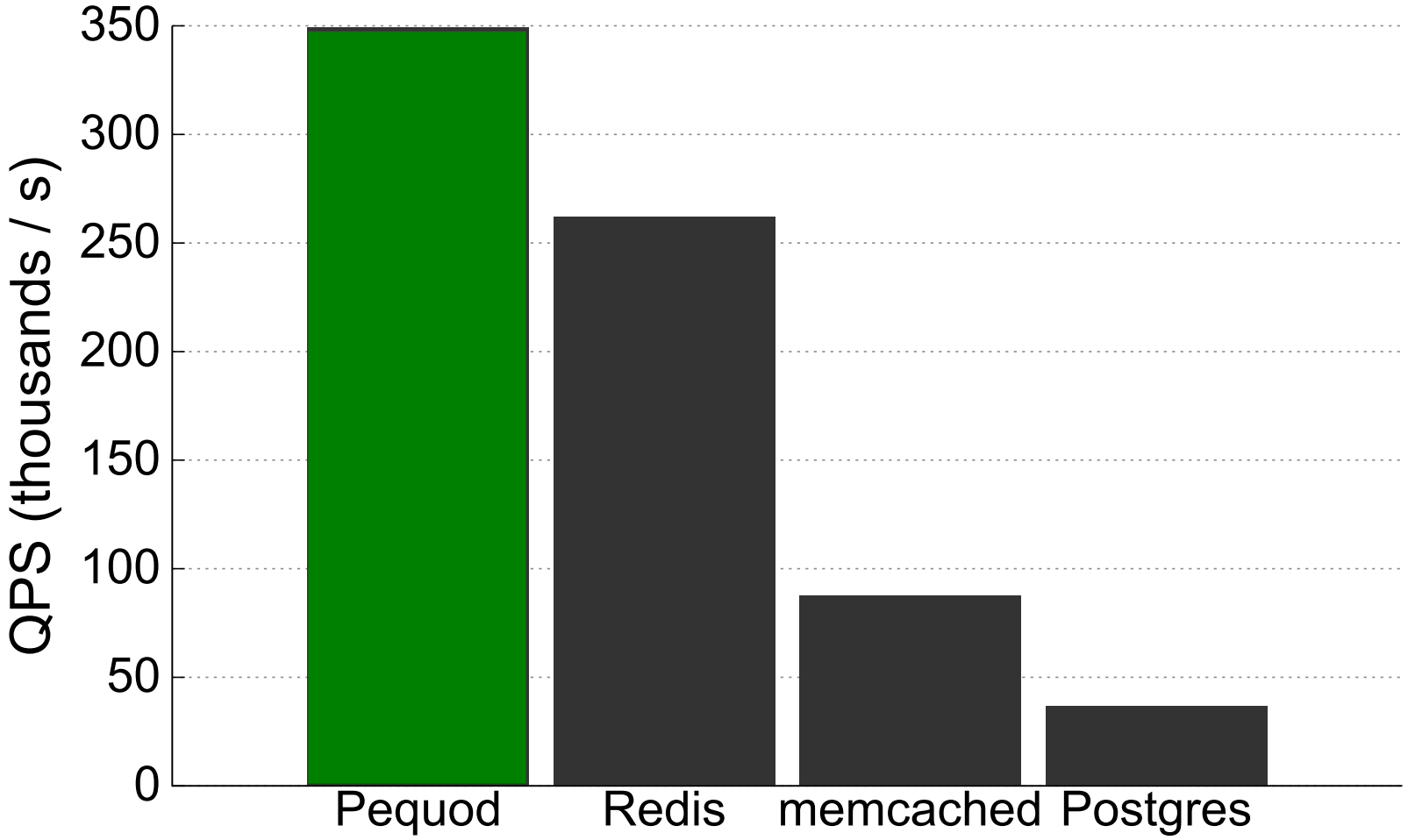
# system comparison

**Do cache joins have key-value cache performance?**

- goal: perform no worse than existing caches

- compare with:
  - fast KV caches: Redis, memcached
  - DB-as-cache: Postgres (in-memory, tuned)
    - Postgres uses "materialized views" (triggers)

# system comparison



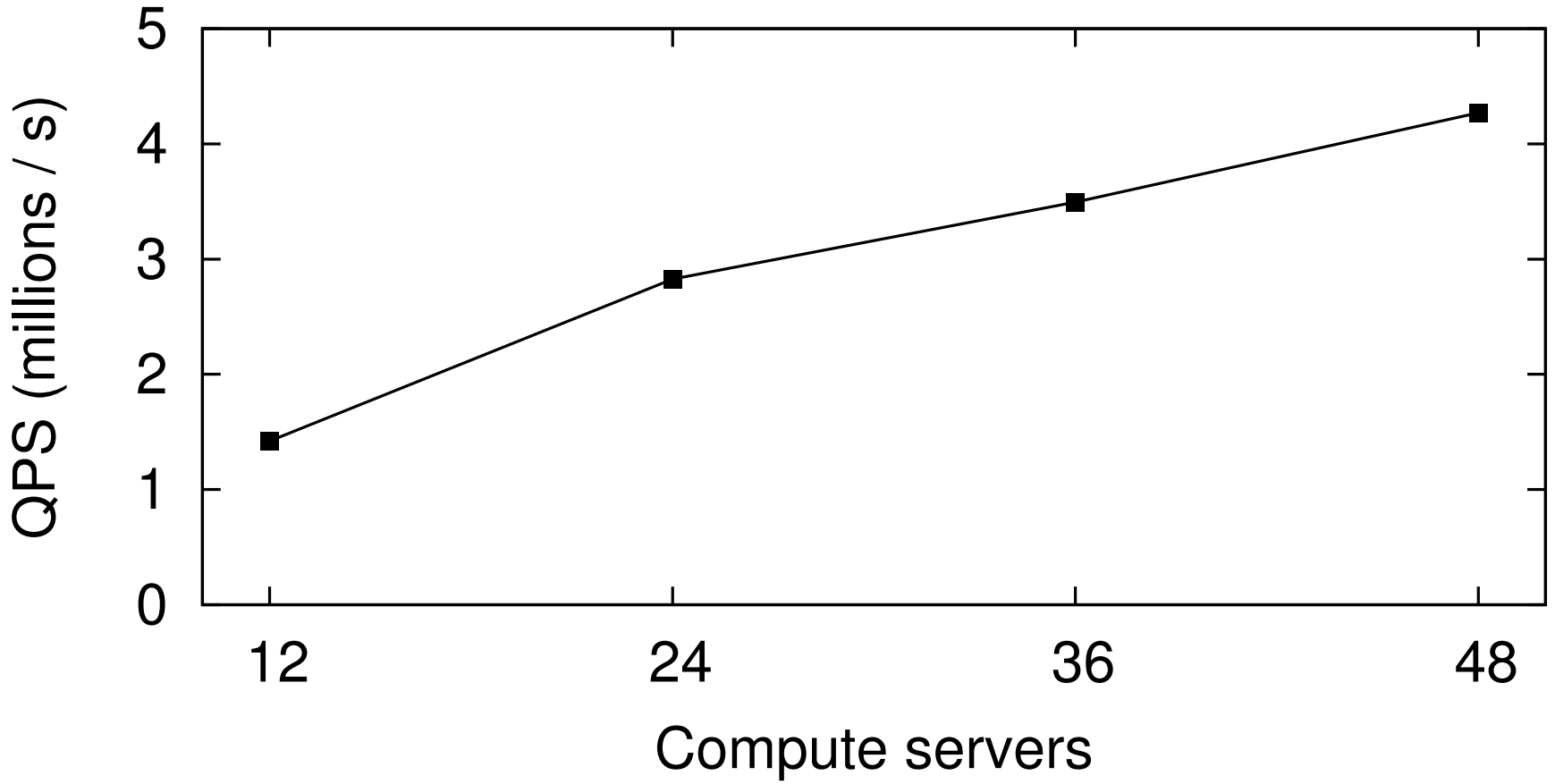
# scaling Pequod

**Will adding servers improve performance?**
**What is the overhead of data movement?**

- cluster on Amazon EC2

- two-tier deployment
  - subscriptions, posts on "base" servers
  - timelines executed on "compute" servers
  - replication is required

# scaling Pequod



# scaling Pequod (overhead)

- steady-state bandwidth for data movement
  - 10 ➔ 16% (larger fanout)

- total memory consumption
  - 290 ➔ 297GB at base (subscription metadata)
  - 1.2 ➔ 1.5TB at compute (duplicate data)

- overhead is noticeable but not crippling

# selected related work

- DMV [Zhou et al, 2007]
  - partial, dynamic database materialized views
- DBProxy [Amiri et al, 2002-3]
  - distributed cache built from databases
  - incremental updates to cached results
- MV in PNUTS [Agrawal et al, 2009]
  - materialized views in a key-value store
  - incremental updates, not partial

# conclusion

- Pequod cache joins
  - programmability of materialized views
  - performance of a key-value cache
  - code release soon! github.com/bryankate