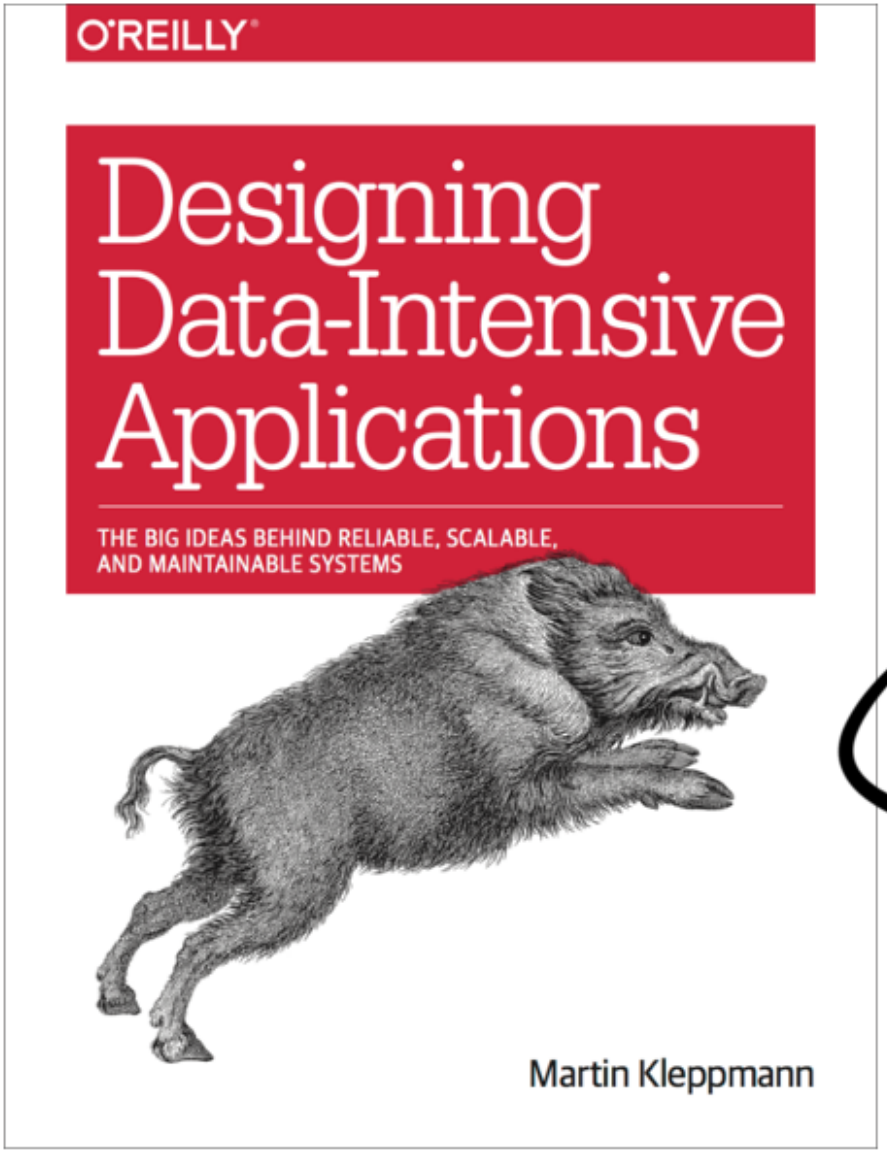
Turning the DB ~ inside-out ~ with samza

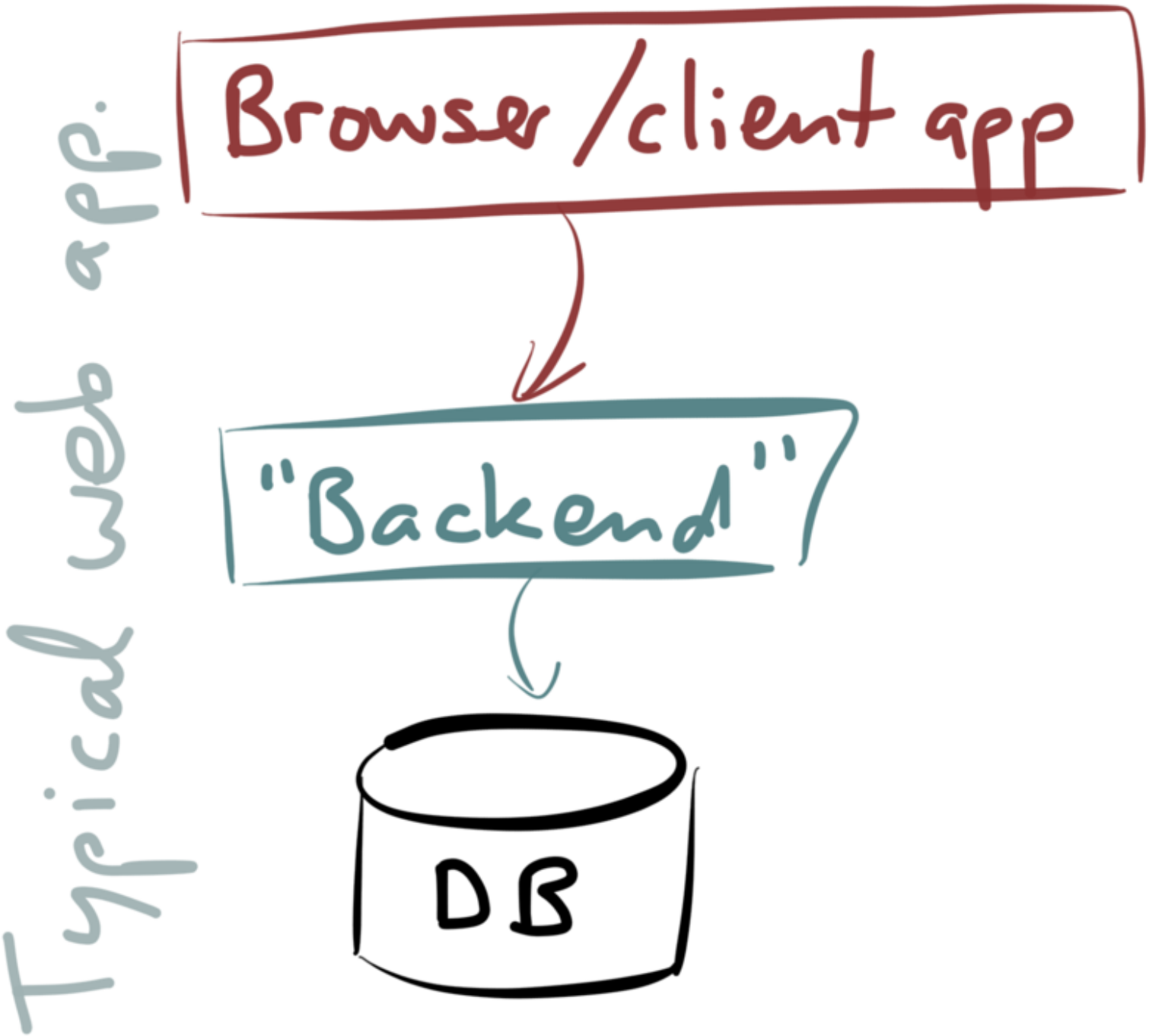
dataintensive.net

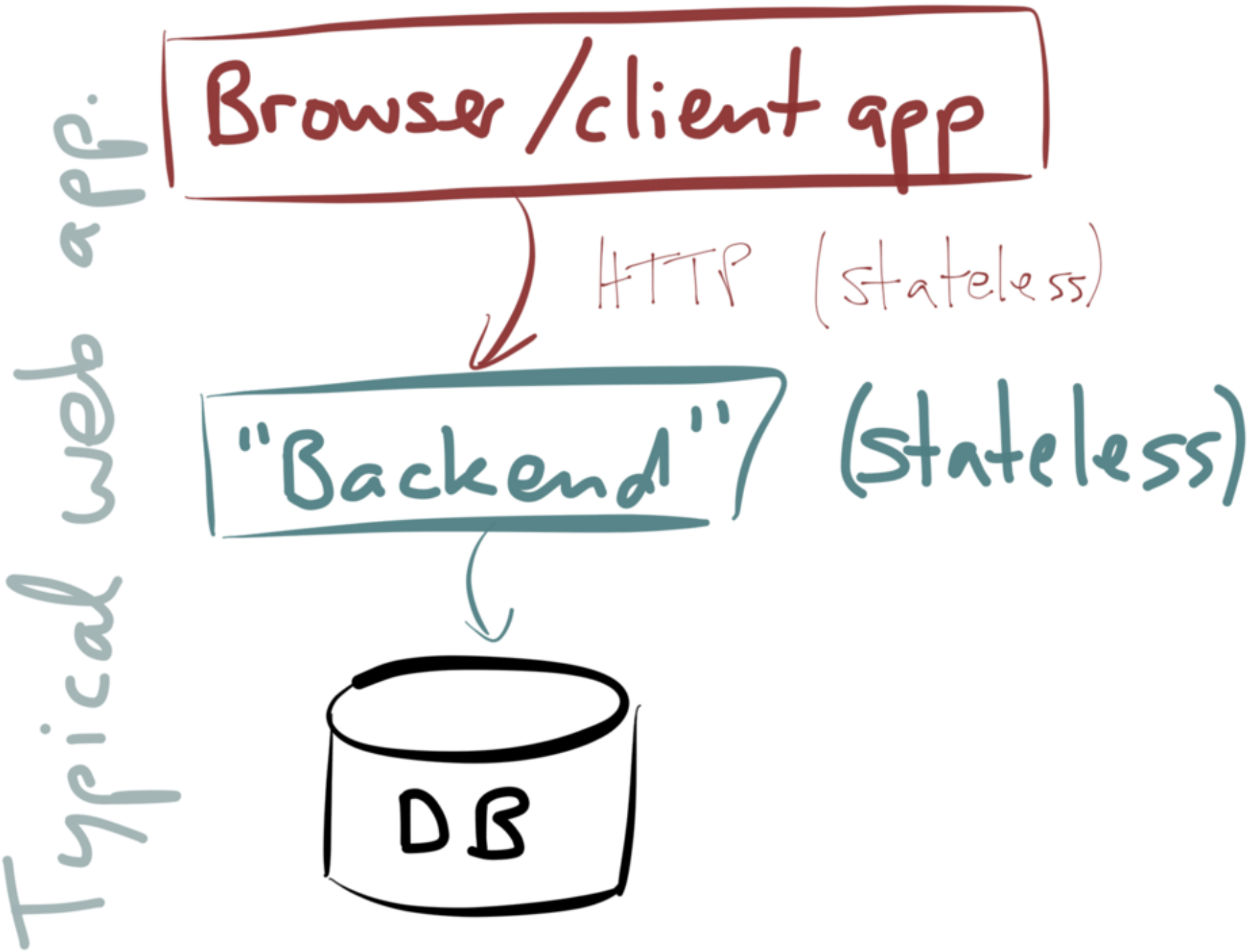


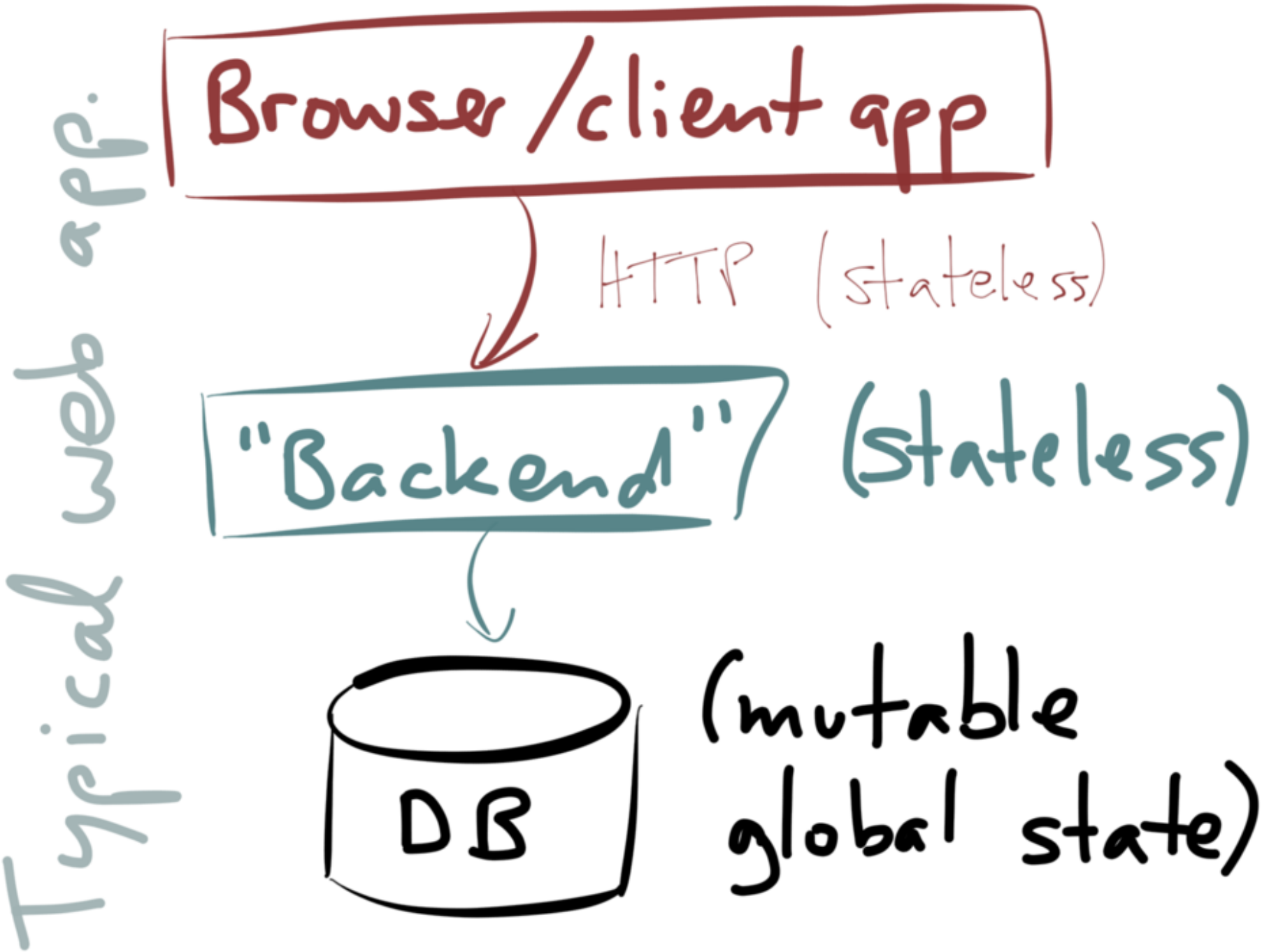
@martinkl

samza.incubator.apache.org

samza


Typical web app.
Browser / client app
"Backend"
DB


Typical web app.
Browser / client app
HTTP (stateless)
"Backend"
(stateless)
DB


Typical web app.
Browser / client app
HTTP (stateless)
"Backend"
(stateless)
DB
(mutable
global state)

"It's always been that way"?


1.
Replication

Shopping cart

customer_id	product_id	quantity
123	888	1
123	999	1
234	444	2
234	555	3
345	666	1

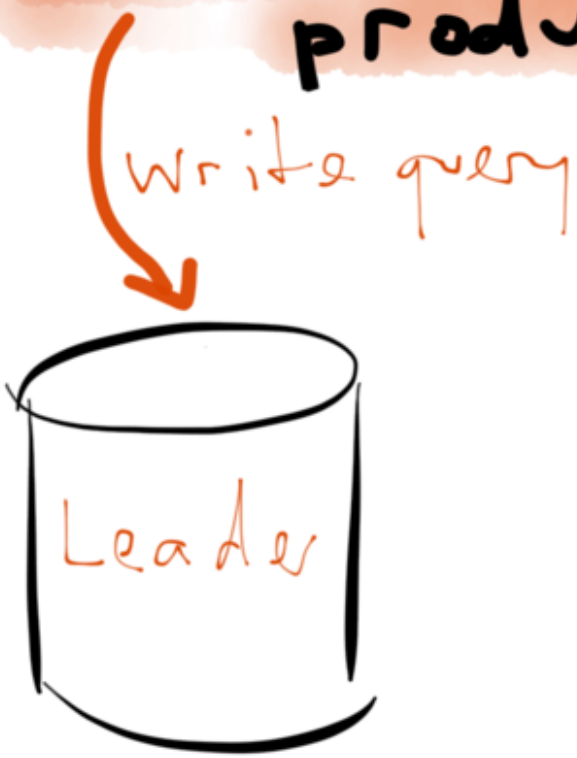
```
update cart set quantity=3
where customer_id=123 and
product_id=999
```

customer_id	product_id	quantity
123	888	1
123	999	1
234	444	2
234	555	3
345	666	1

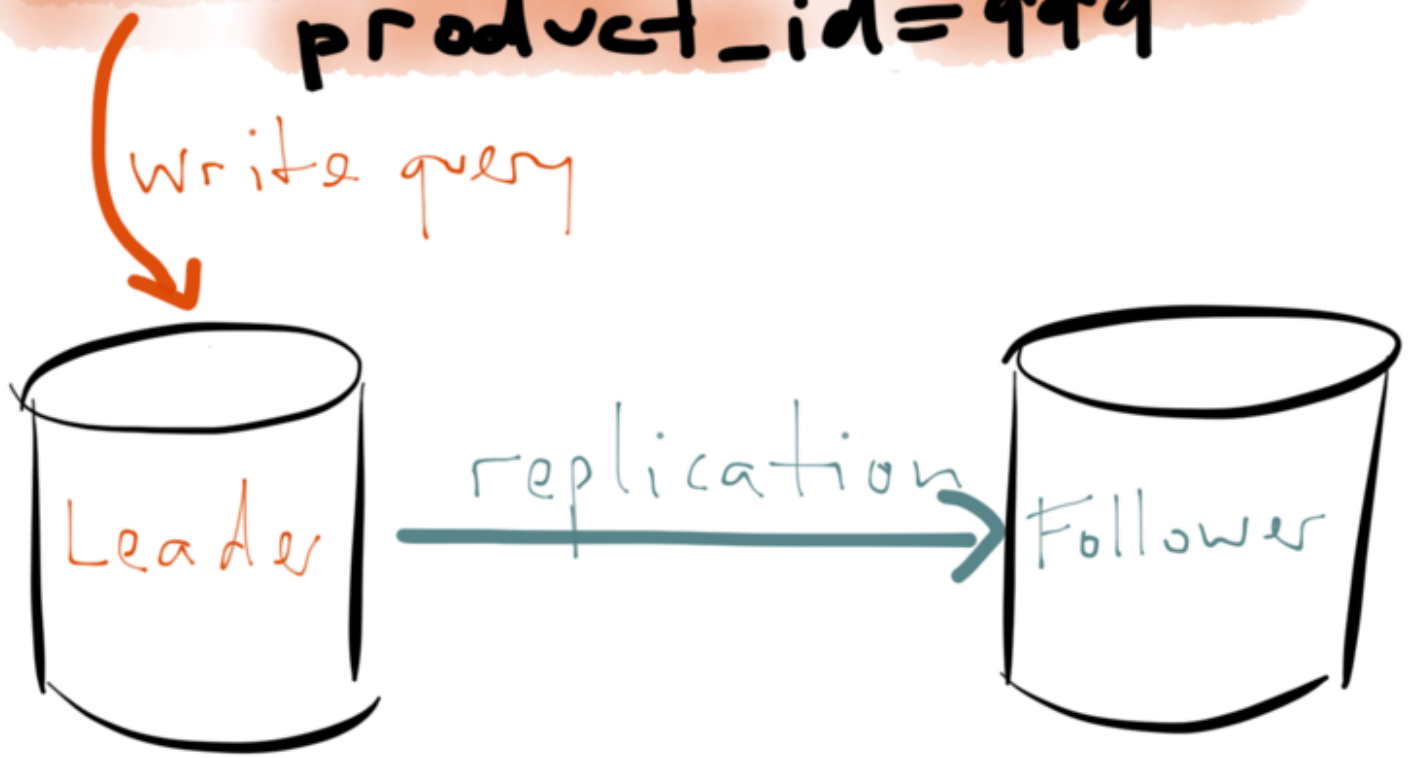
```
update cart set quantity=3
where customer_id=123 and
product_id=999
```

customer_id	product_id	quantity
123	888	1
123	999	3
234	444	2
234	555	3
345	666	1

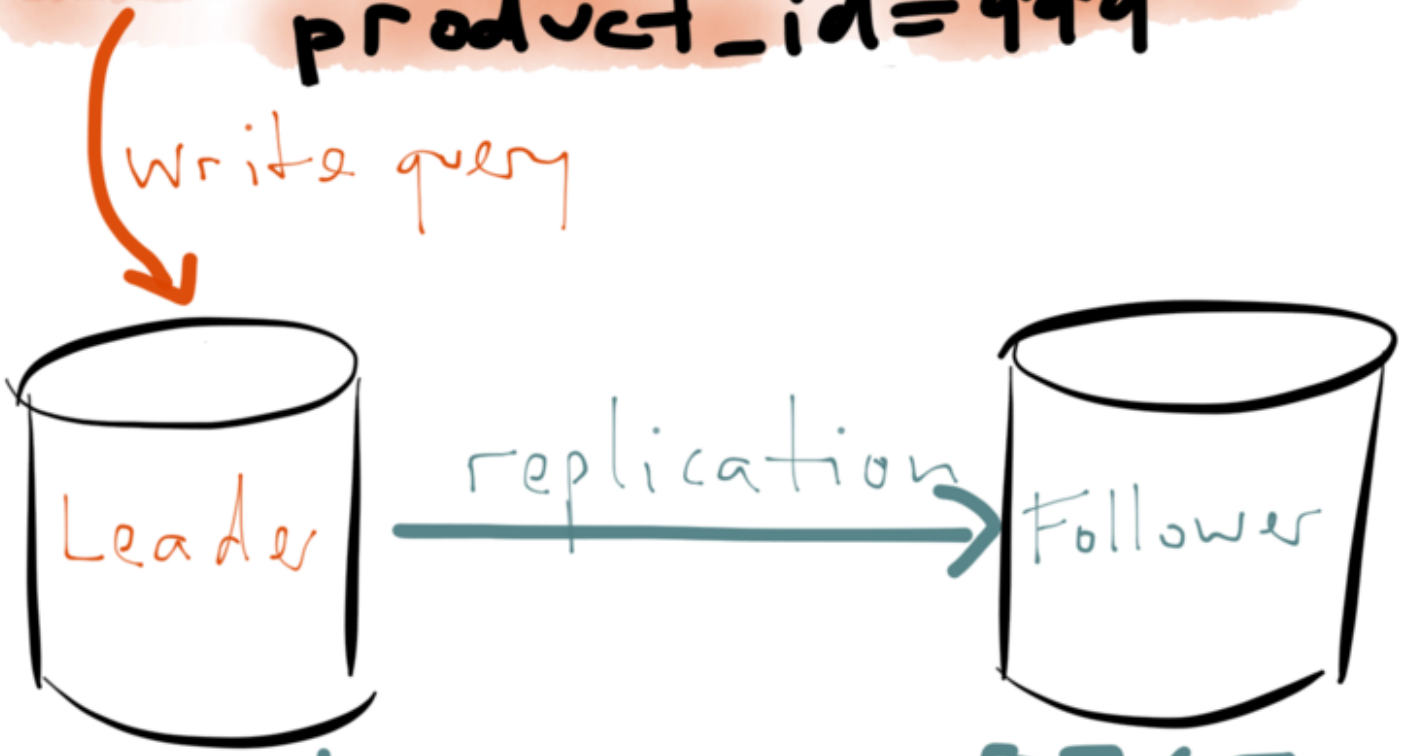
```
update cart set quantity = 3
where customer_id = 123 and
      product_id = 999
```



```
update cart set quantity=3
where customer_id=123 and
      product_id=999
```



update cart set quantity = 3
where customer_id = 123 and
product_id = 999



change row 8765
old = [123, 999, 1]
new = [123, 999, 3]

```
update cart set quantity=3
where customer_id=123 and
product_id=999
```

VS.

"at 2014-09-18 14:10:59,
customer 123 changed quantity of
product 999 in their cart
from 1 to 3"

```
update cart set quantity = 3
where customer_id = 123 and
      product_id = 999
```

(imperative state mutation)

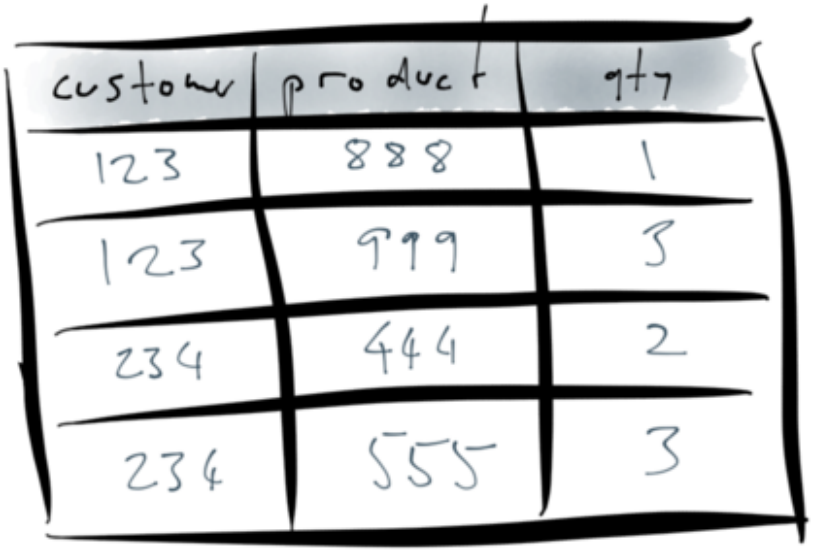
vs.

"at **2014-09-18 14:10:59**,
customer **123** changed quantity of
product **999** in their cart
from **1** to **3**"

(immutable event / fact)

Secondary indexes

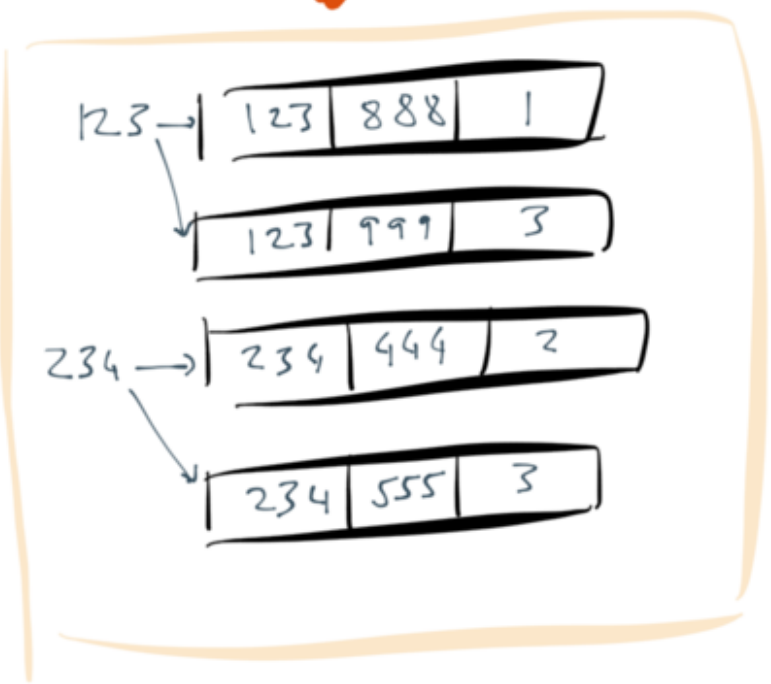
customer	product	qty
123	888	1
123	999	3
234	444	2
234	555	3

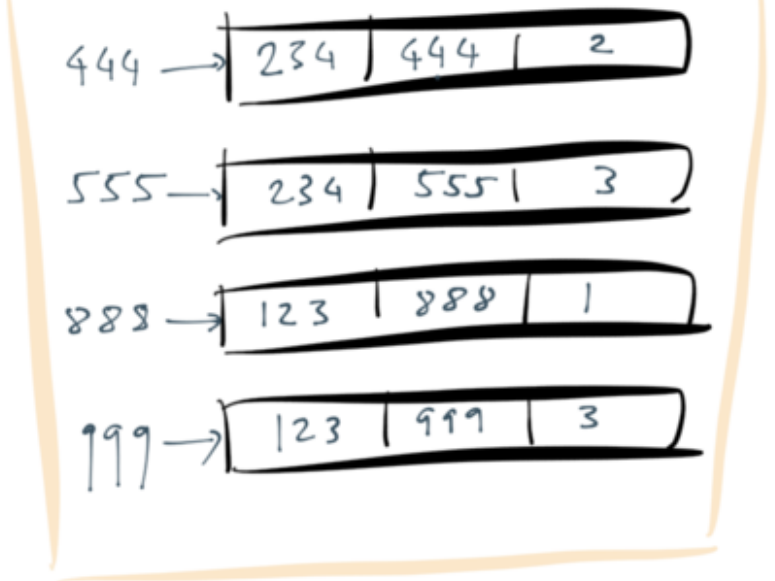


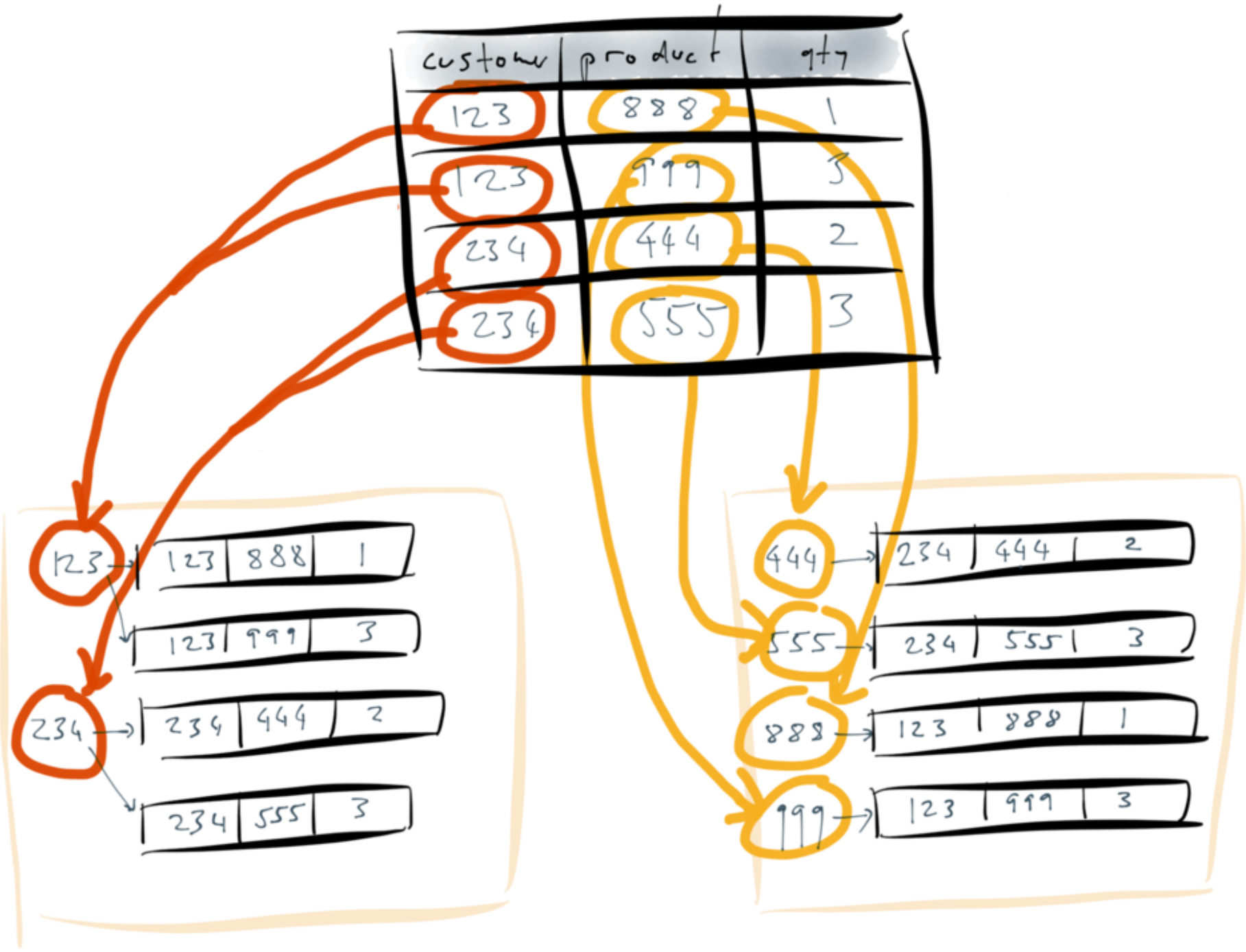
customer	product	qty
123	888	1
123	999	3
234	444	2
234	555	3

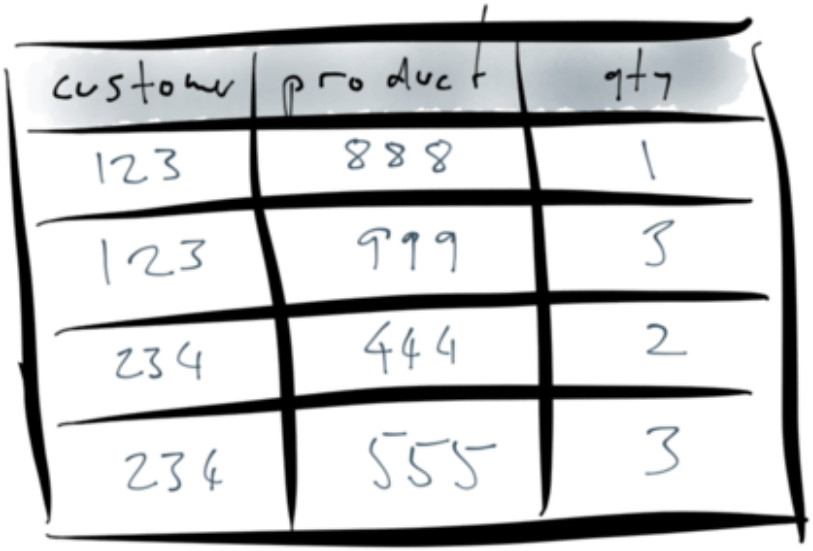
```
CREATE INDEX ON
    cart (customer_id);
CREATE INDEX ON
    cart (product_id);
```

customer	product	qty
123	888	1
123	999	3
234	444	2
234	555	3






customer
product
qty
123
888
1
123
999
3
234
444
2
234
555
3
123
123 888 1
123 999 3
234
234 444 2
234 555 3
444
234 444 2
555
234 555 3
888
123 888 1
999
123 999 3

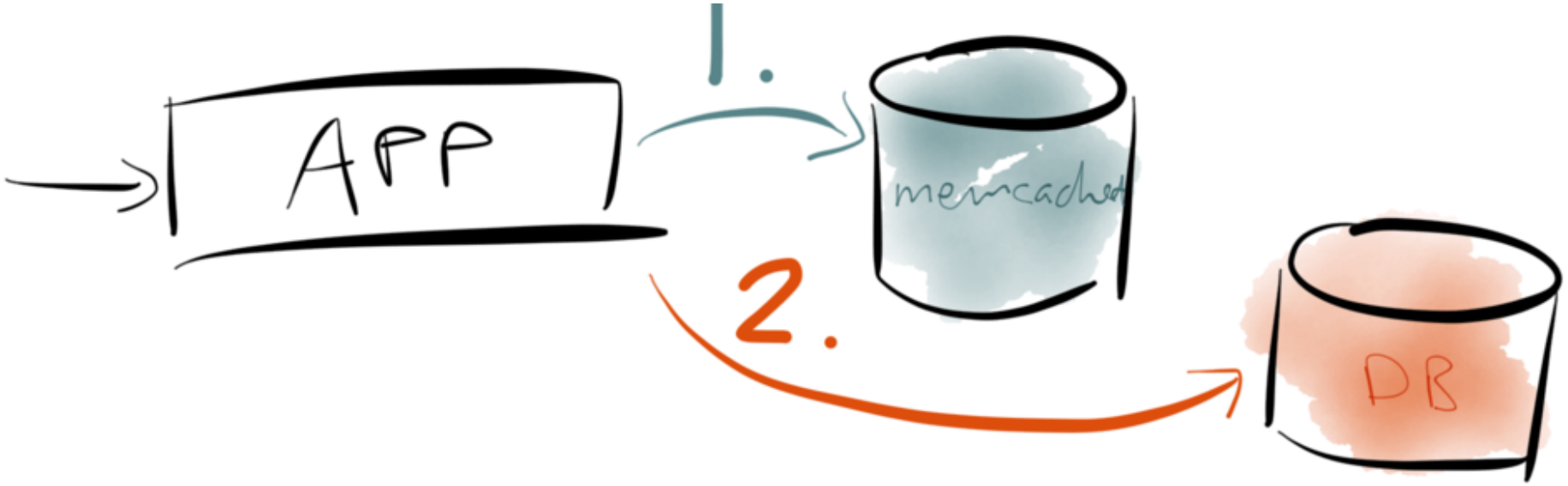


CREATE INDEX CONCURRENTLY...

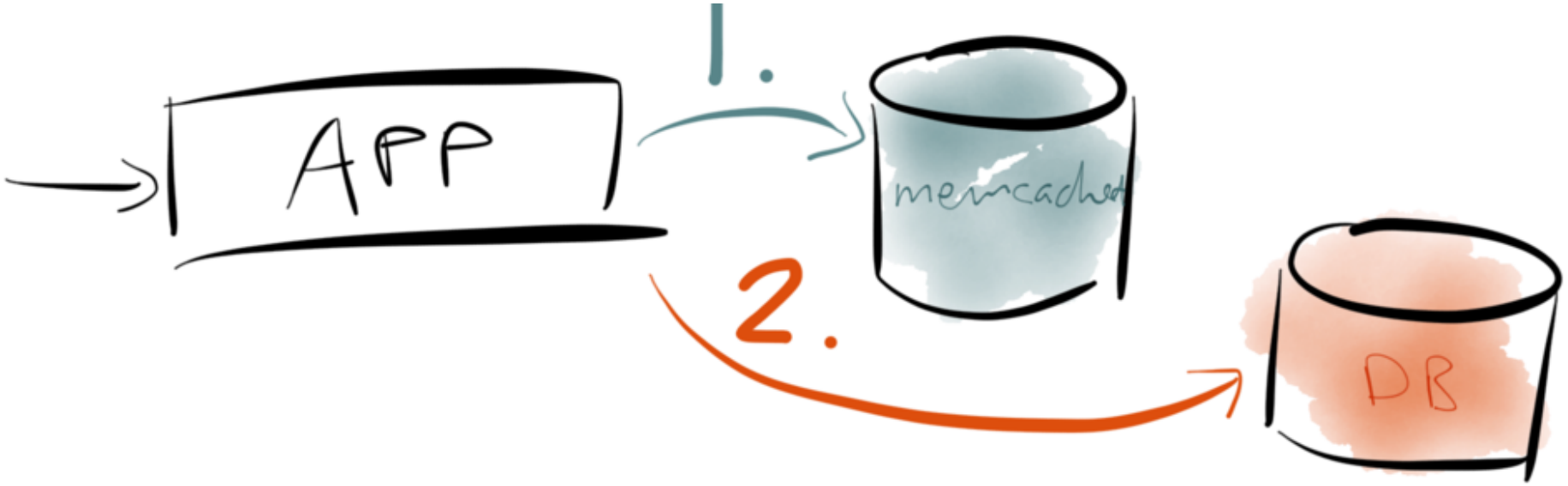

3.
Caching


```
r = cache.get(key)
```

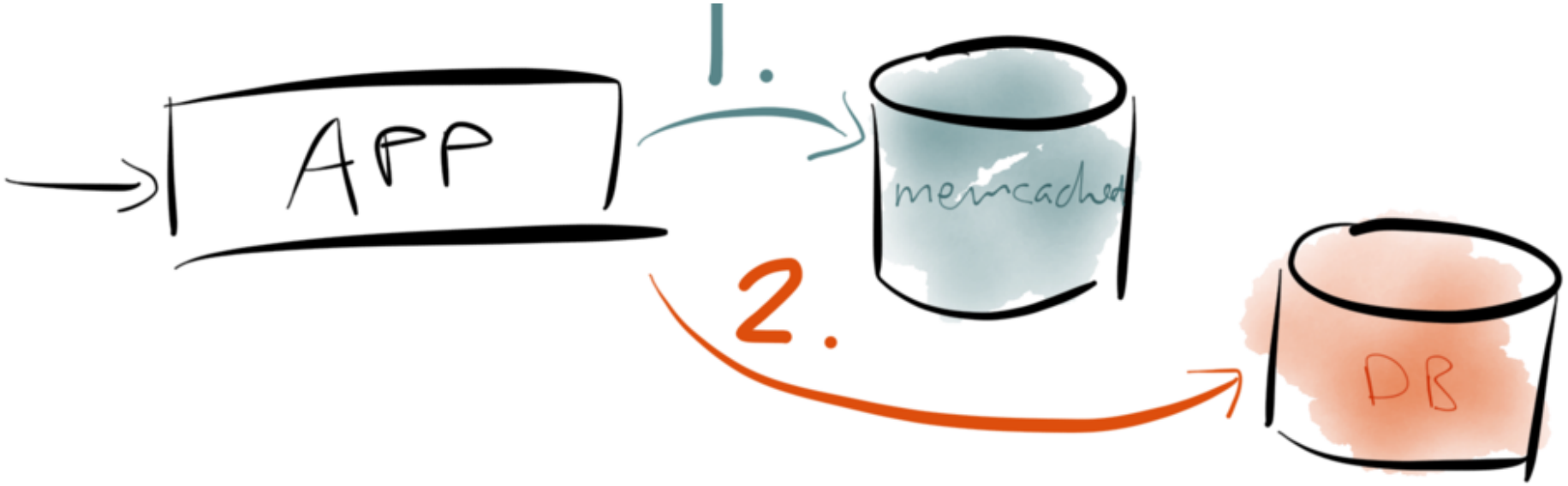

```
r = cache.get(key)
if (!r) {
  r = db.get(key)
  cache.put(key, r)
}
return r;
```



- invalidation?



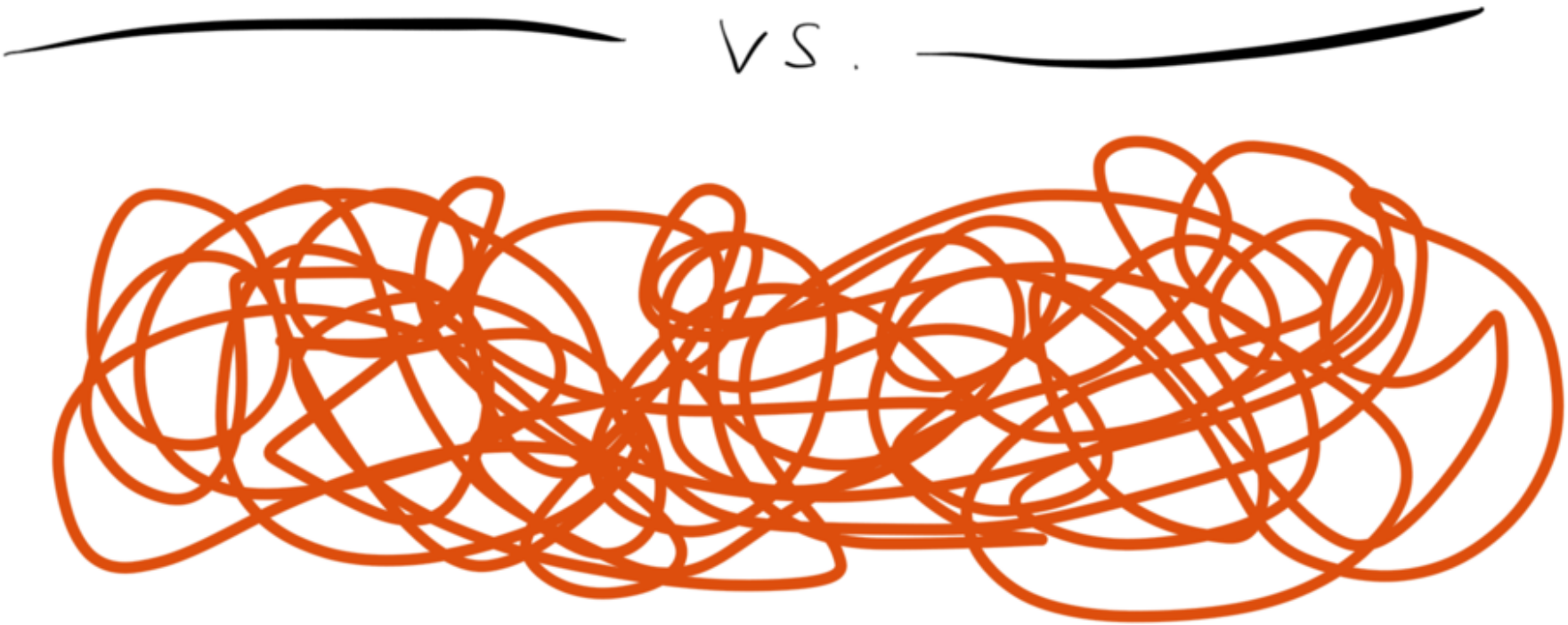
- invalidation?
- race conditions / consistency issues?



- invalidation?
- race conditions / consistency issues?
- cold start / bootstrapping

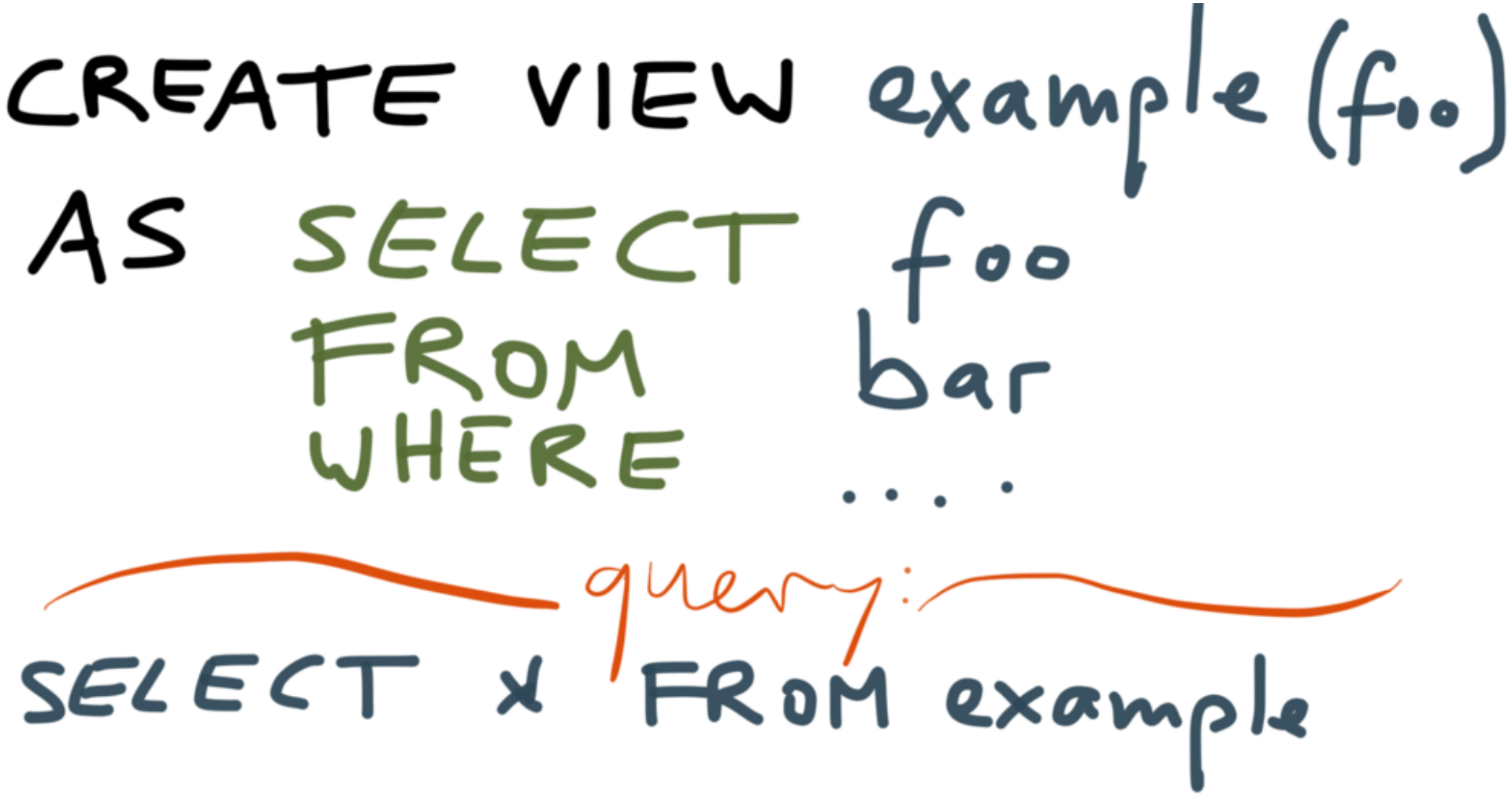
CREATE INDEX...

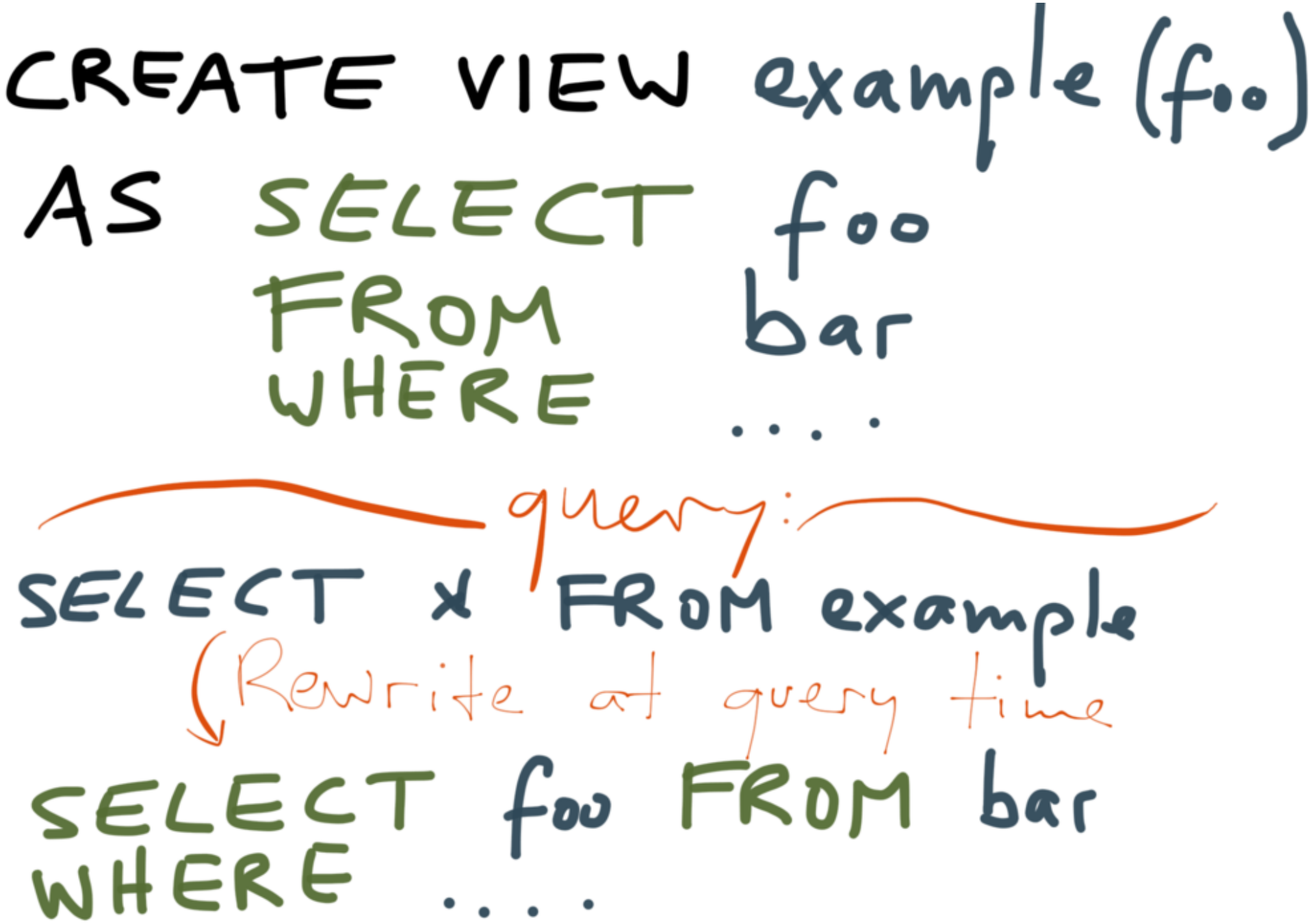
(1 line of SQL!)



Materialized Views

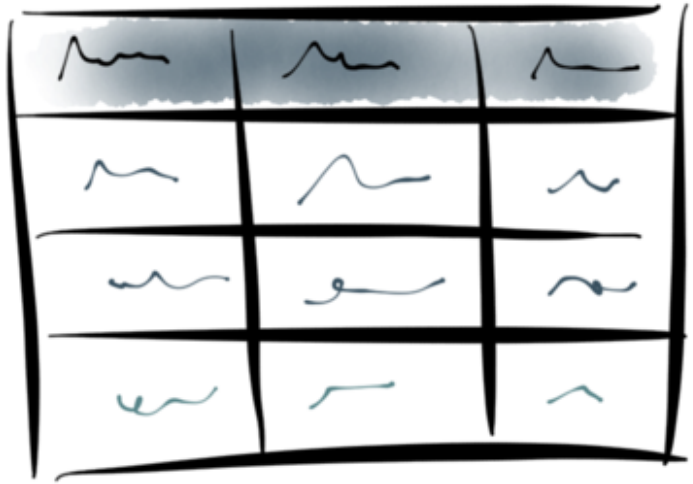
```
CREATE VIEW example(foo)
AS SELECT foo
   FROM bar
   WHERE ...
```


CREATE VIEW example (foo)
AS SELECT foo
FROM bar
WHERE ....
query:
SELECT x FROM example


CREATE VIEW example (foo)
AS SELECT foo
FROM bar
WHERE ....
query:
SELECT x FROM example
Rewrite at query time
SELECT foo FROM bar
WHERE ....

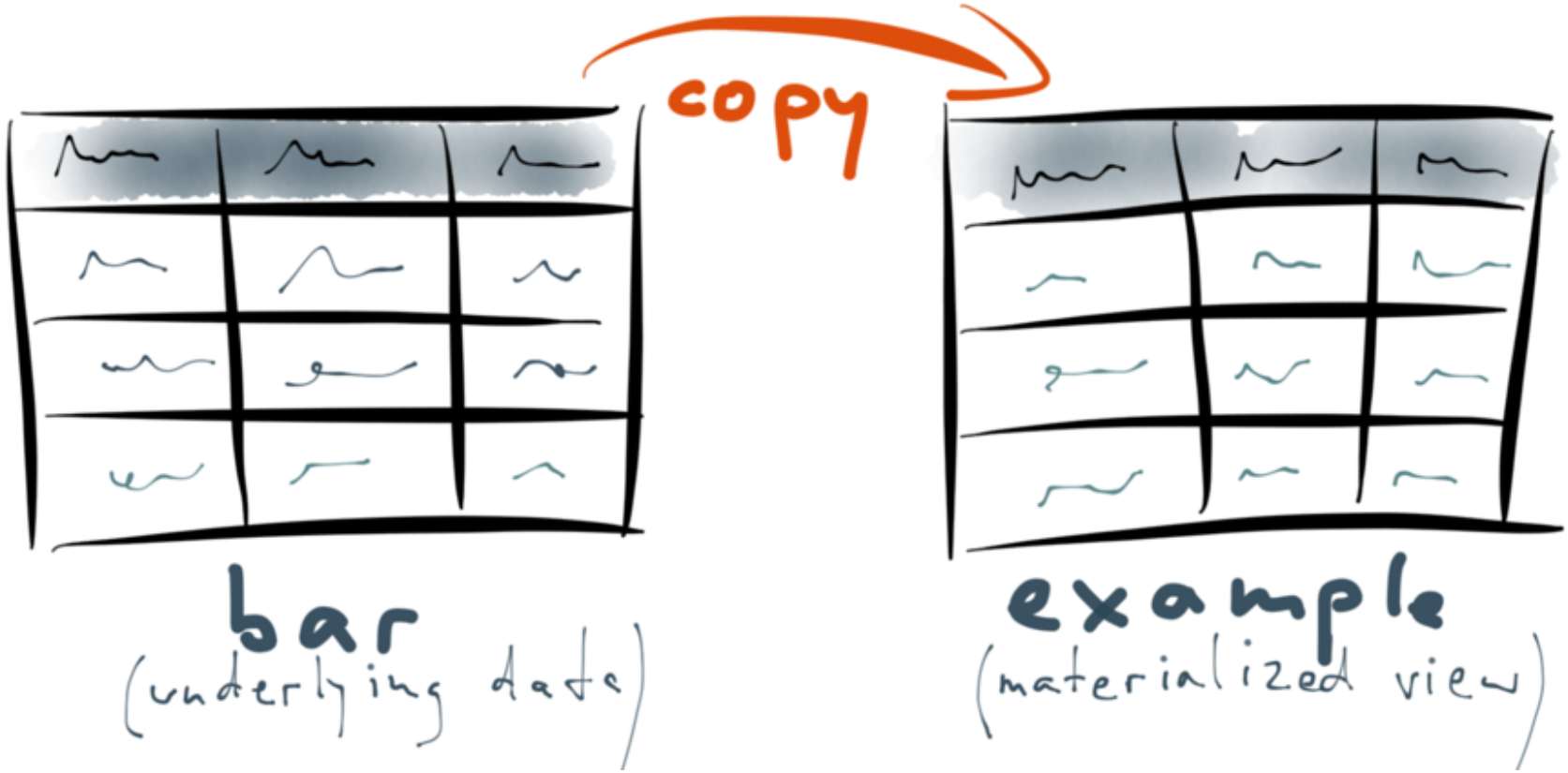
```
CREATE MATERIALIZED
  VIEW example (foo)
AS SELECT foo FROM bar ...
```

CREATE MATERIALIZED VIEW example (foo) AS SELECT foo FROM bar ...



bar

CREATE MATERIALIZED VIEW example (foo) AS SELECT foo FROM bar ...



1. Replication
2. Secondary indexing
3. Caching
4. Materialized views

1. Replication
2. Secondary indexing
3. Caching
4. Materialized views

DERIVED DATA

1. Replication

2. Secondary indexing

3. Caching

4. Materialized views

DERIVED DATA

1. Replication

2. Secondary indexing

3. Caching

4. Materialized views

DERIVED DATA

1. Replication 
2. Secondary indexing 
3. Caching 
4. Materialized views

DERIVED DATA

1. Replication

2. Secondary indexing

3. Caching

4. Materialized views

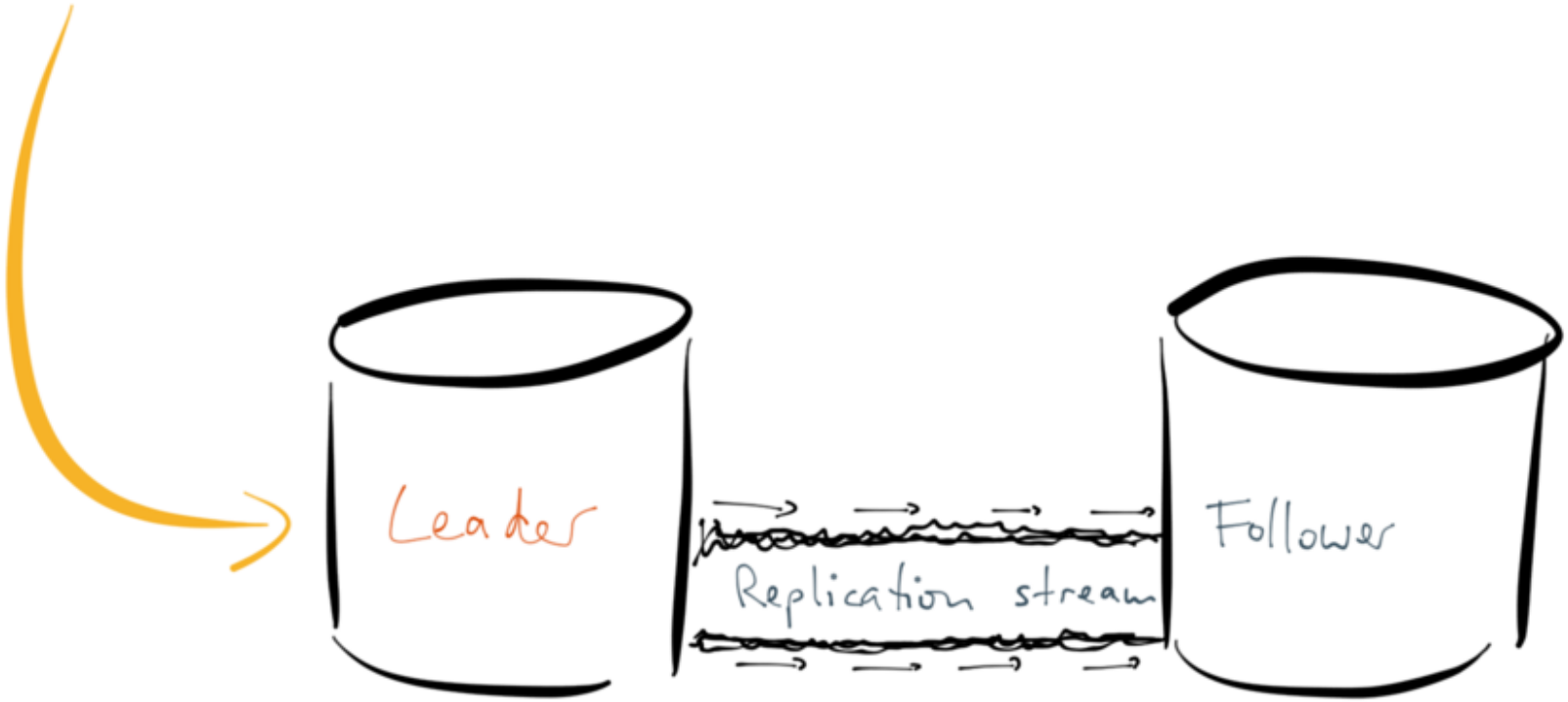
DERIVED DATA

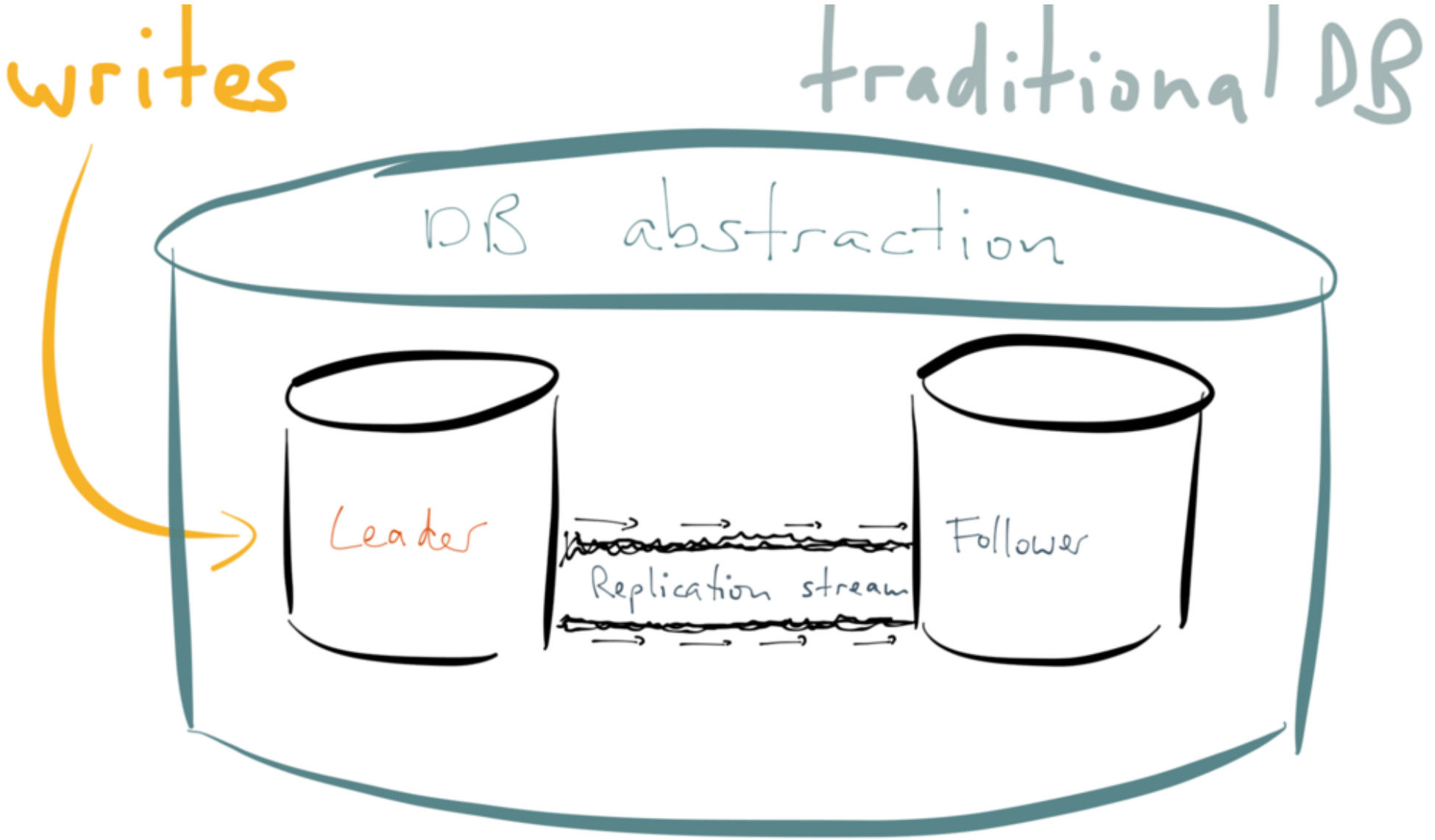

Magically
self-updating
cache...

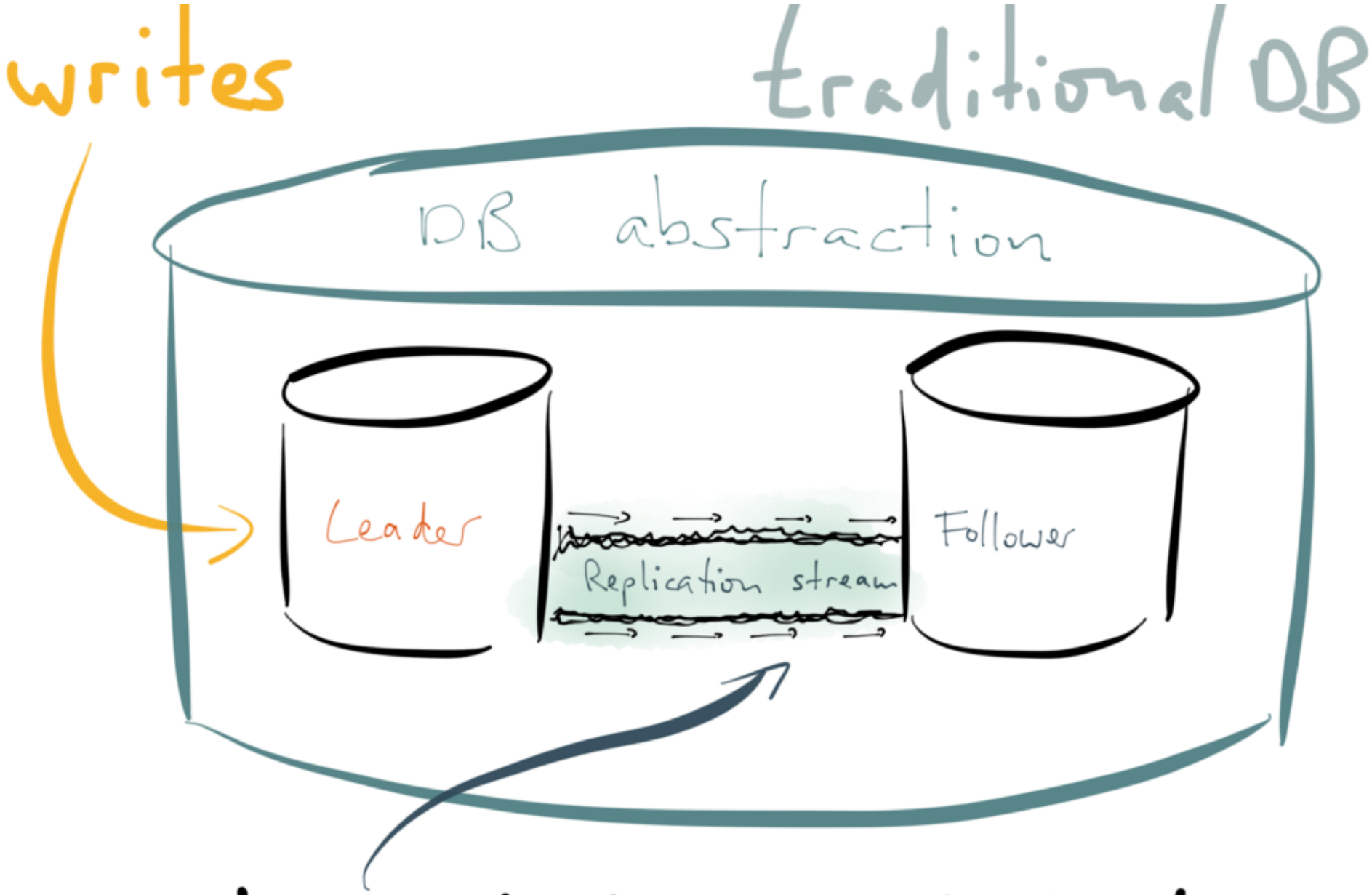

Let's rethink
materialized views!

writes

traditional DB



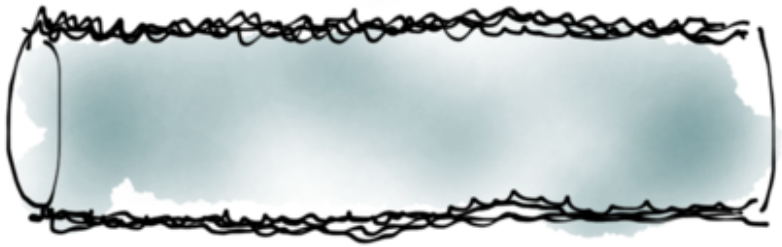

writes
traditional DB
DB abstraction
Leader
Replication stream
Follower



implementation detail.

"unbundled" DB

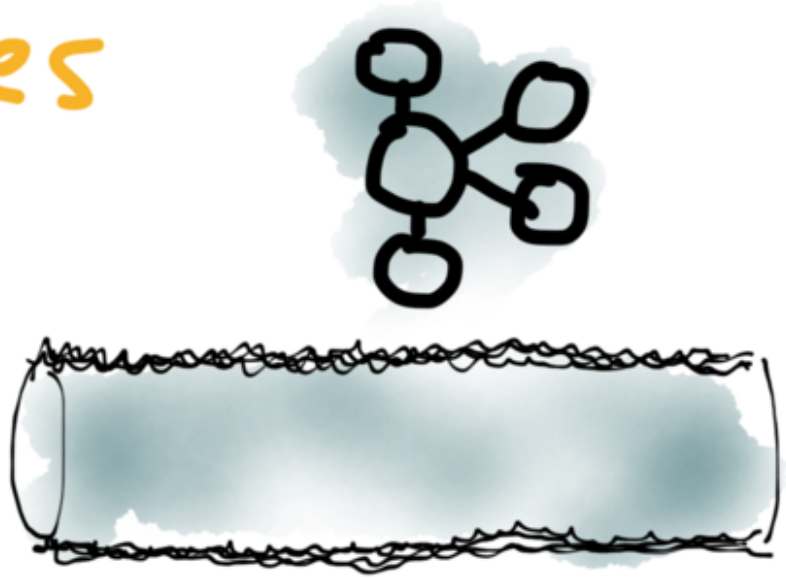
writes



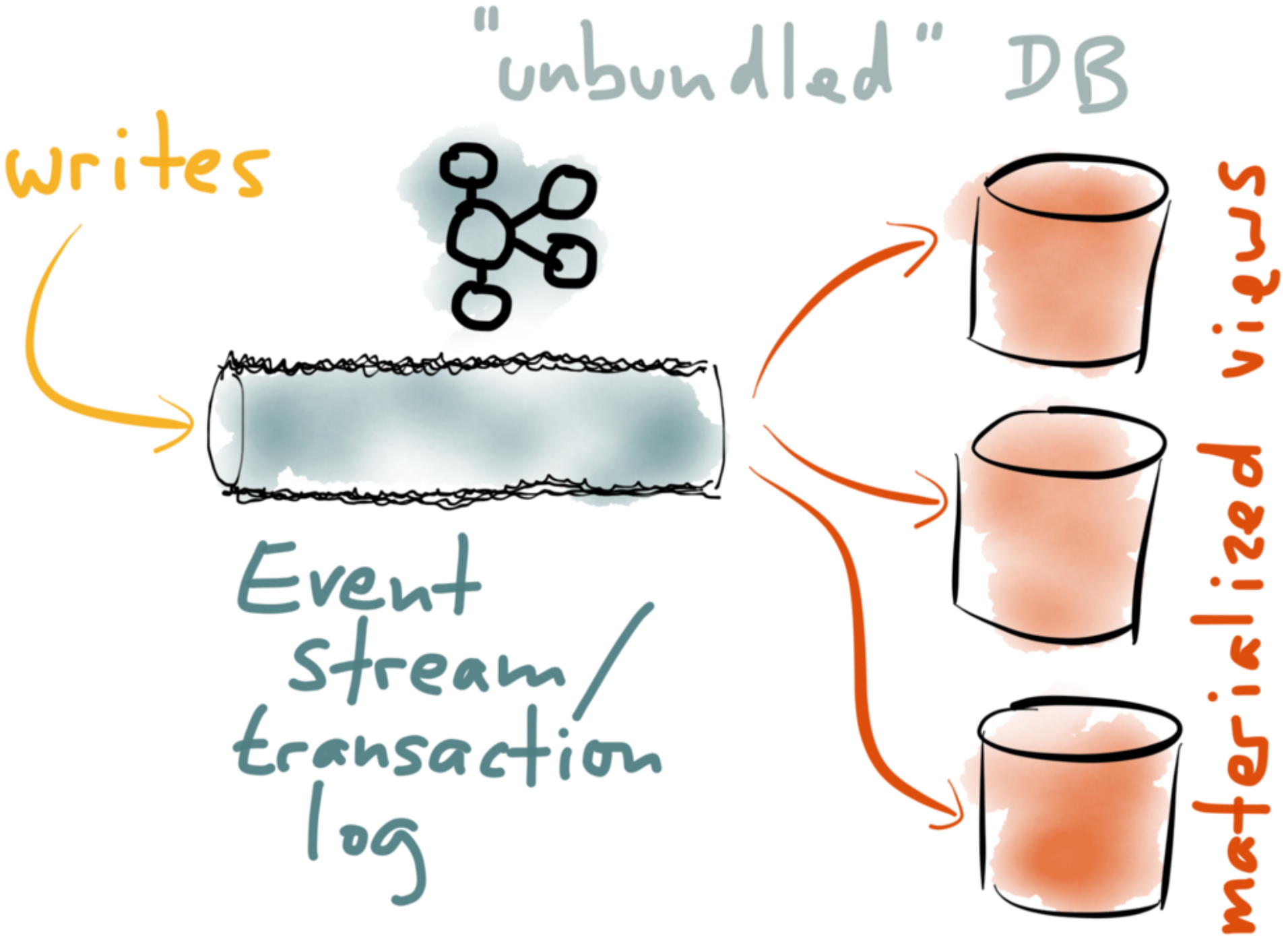
Event stream / transaction log

"unbundled" DB

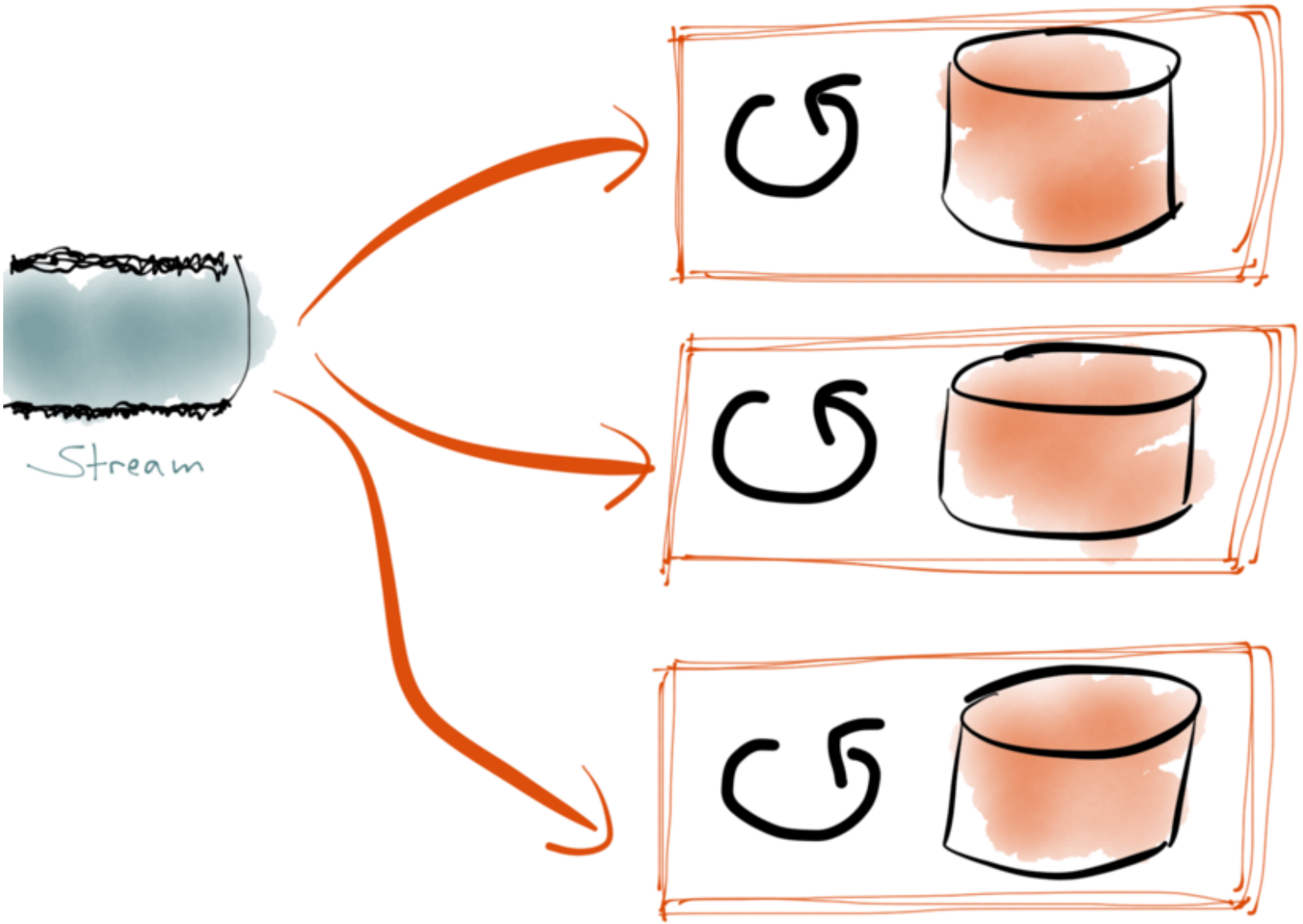
writes



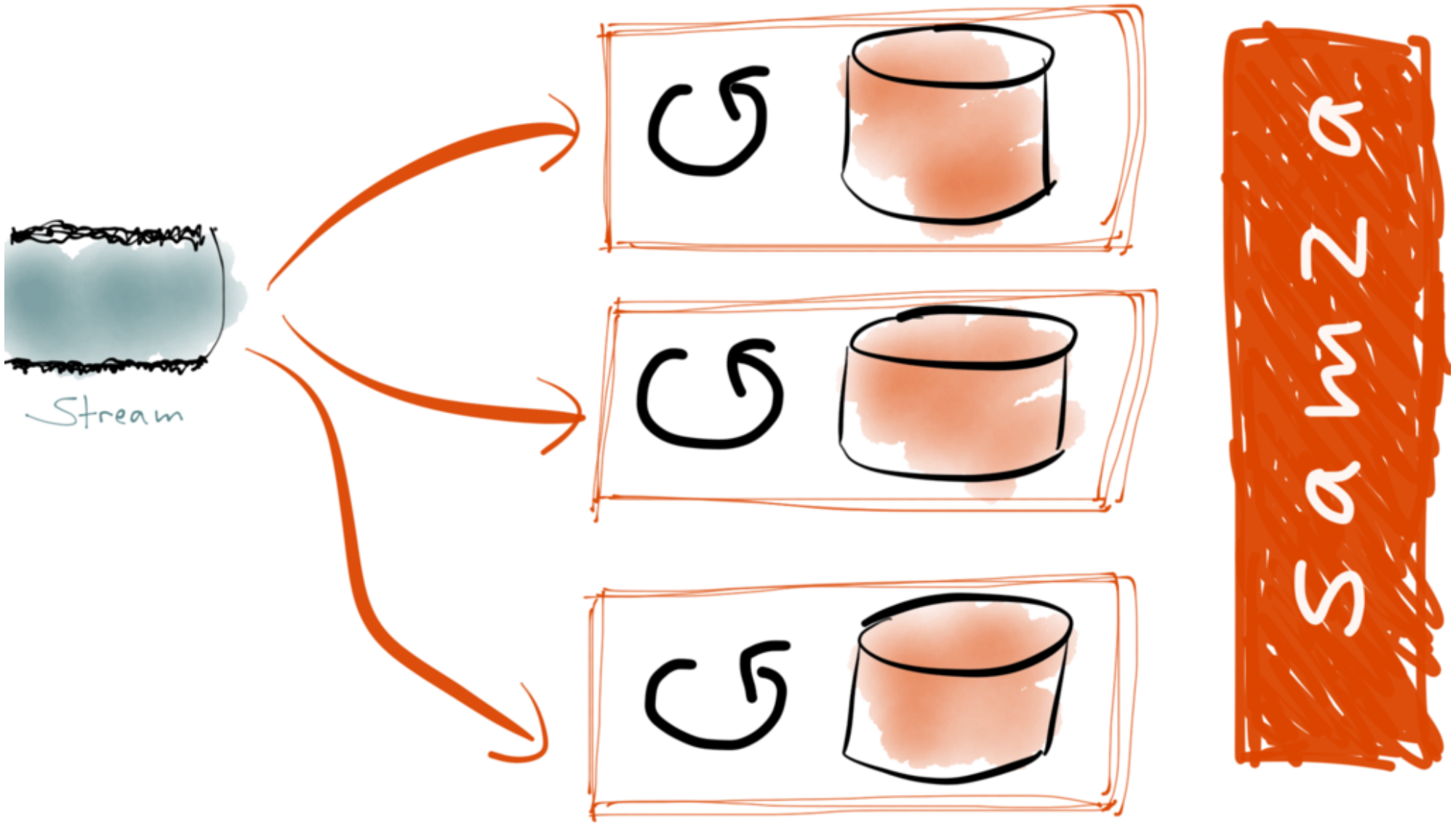
Event stream / transaction log


"unbundled" DB
writes
Event stream / transaction log
materialized views

"unbundled" DB



"unbundled" DB



Writes:

~~mutate state in place~~

Writes:

~~mutate state in place~~

append-only stream of immutable facts

— simple, scalable, fault-tolerant, fast sequential I/O

(see also: Datalog, Datomic, Lambda Architecture)

Writes:

~~mutate state in place~~

append-only stream of immutable facts

— simple, scalable, fault-tolerant, fast sequential I/O

(see also: Datalog, Datomic, Lambda Architecture)

→ Apache Kafka

Reads:

~~query the database~~

Reads:

~~query the database~~

- consume & join streams
- maintain materialized views

Reads:

~~query the database~~

- consume & join streams
- maintain materialized views

→ Apache Samza

Cache vs. materialized view:

~~read-through cache~~
~~updated by app~~

Cache vs. materialized view:

~~read-through cache~~
~~updated by app~~

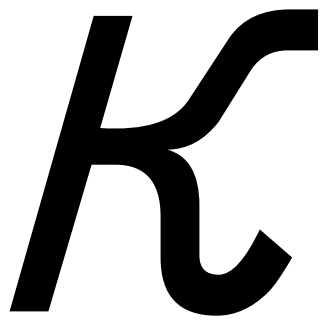
- materialized view = precomputed

Cache vs. materialized view:

~~read-through cache~~
~~updated by app~~

- materialized view = precomputed
- to build new view, consume input since beginning of time

(just like building a new DB index)



Mechanics:

deployment monitoring
cluster operations integrations
fault-tolerance performance
security onboarding isolation
tooling documentation support

...being worked on (contributions welcome!)


Why?

1. Better Data

1. Better Data

- Good for analytics
- Separation of concerns between writing and reading
- Write once, read from many different views
- Historical point-in-time queries
- Recovery from human error (eg. bad code deployed)

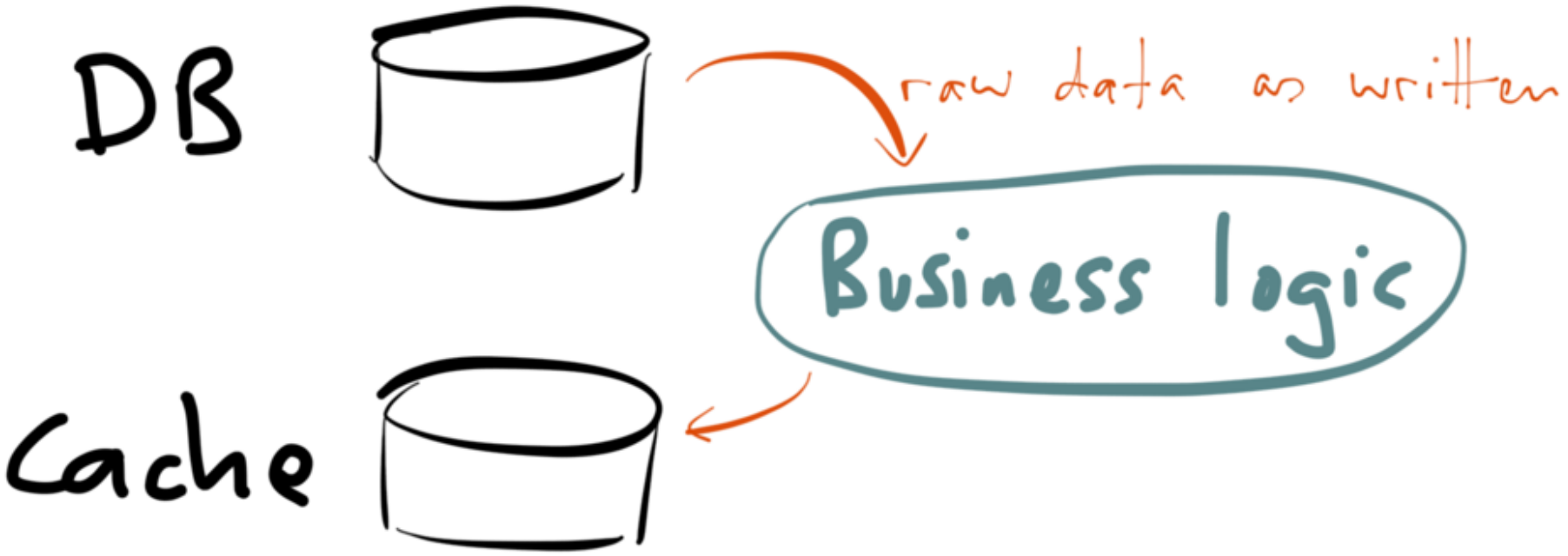
2. Fully precomputed caches

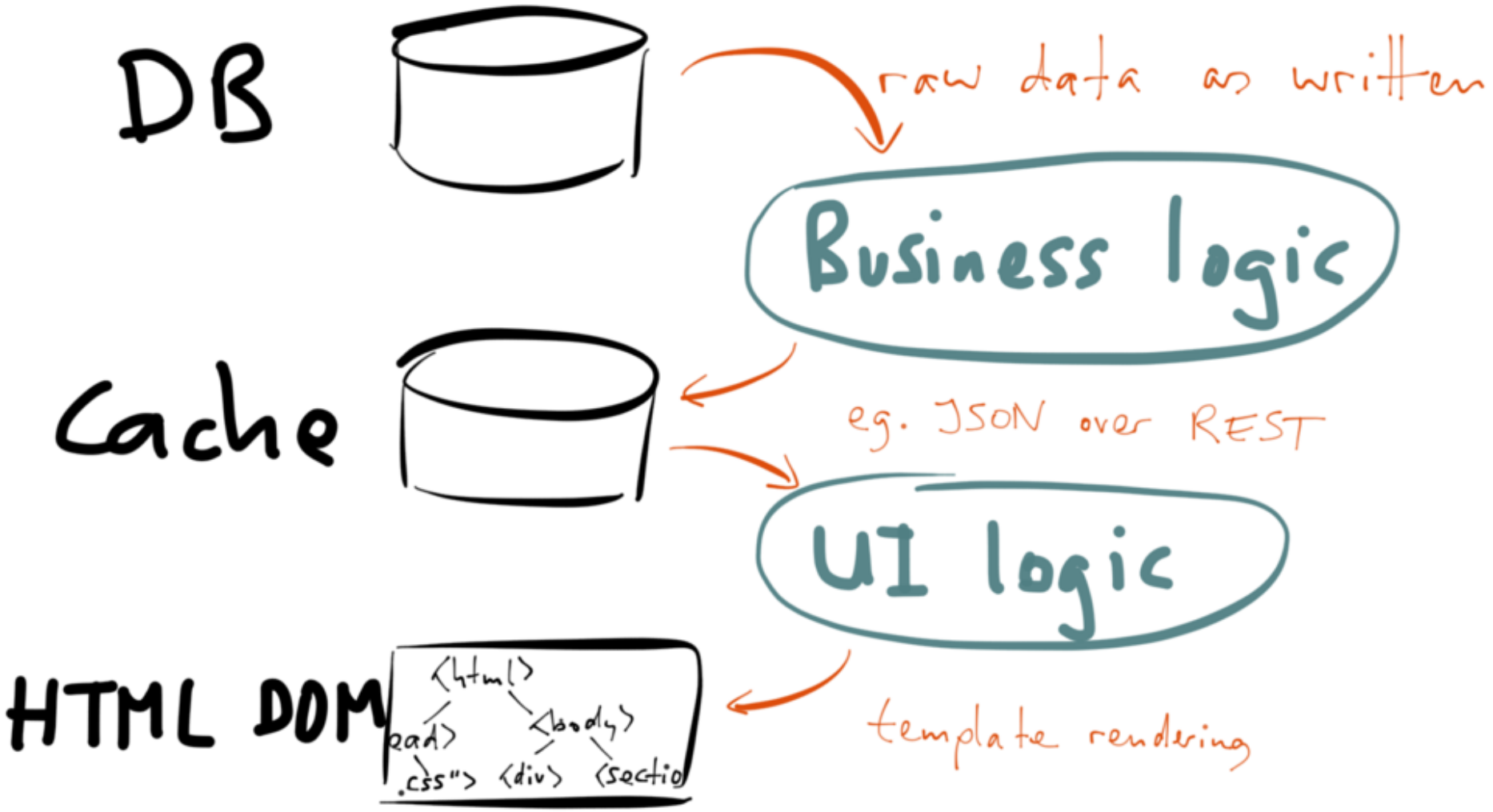
② Fully precomputed caches

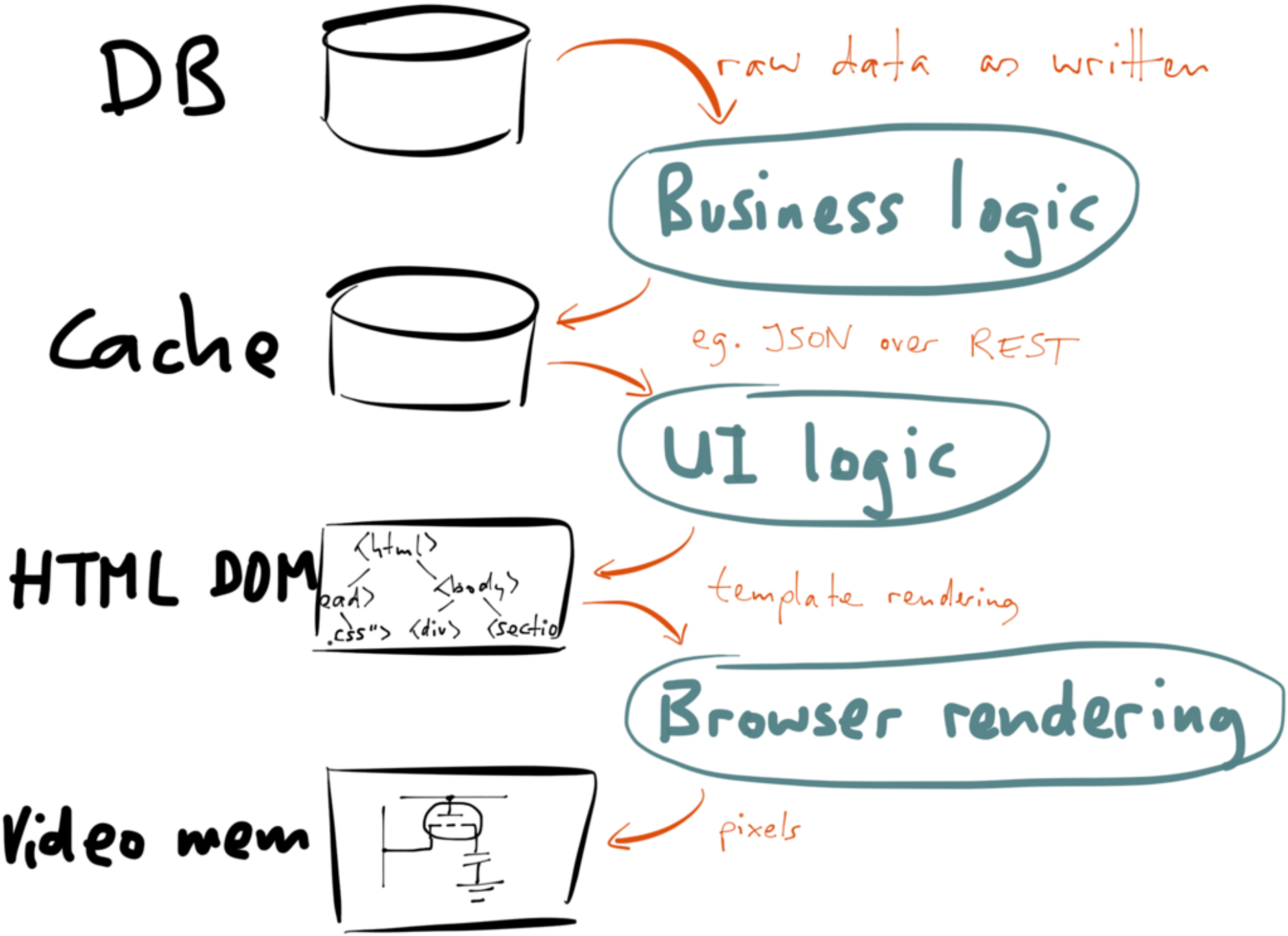
No cold cache/warming, no hit/miss rate, no race conditions, no complex invalidation logic, better isolation/robustness, etc ...

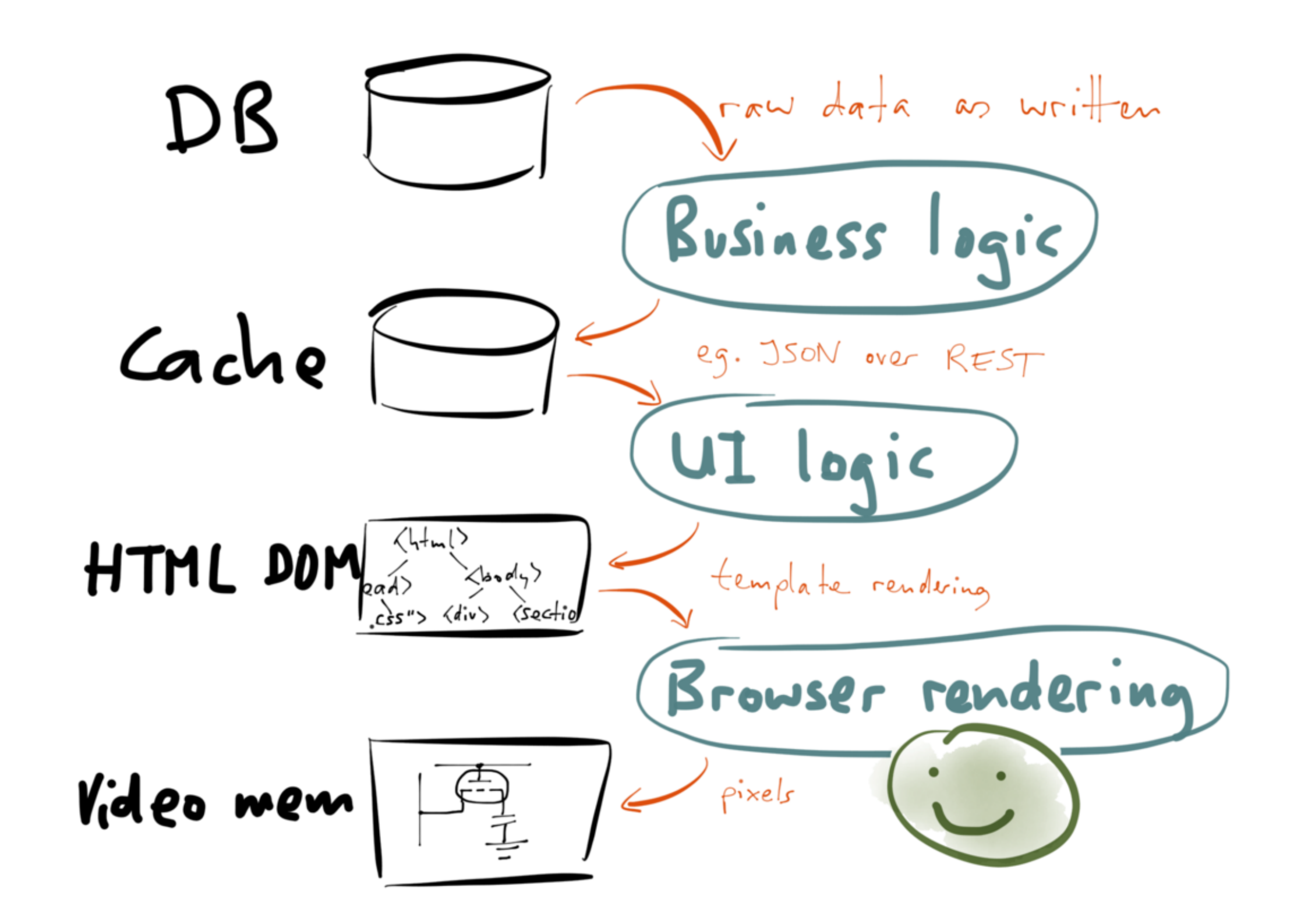

3. Streams
everywhere!

DB

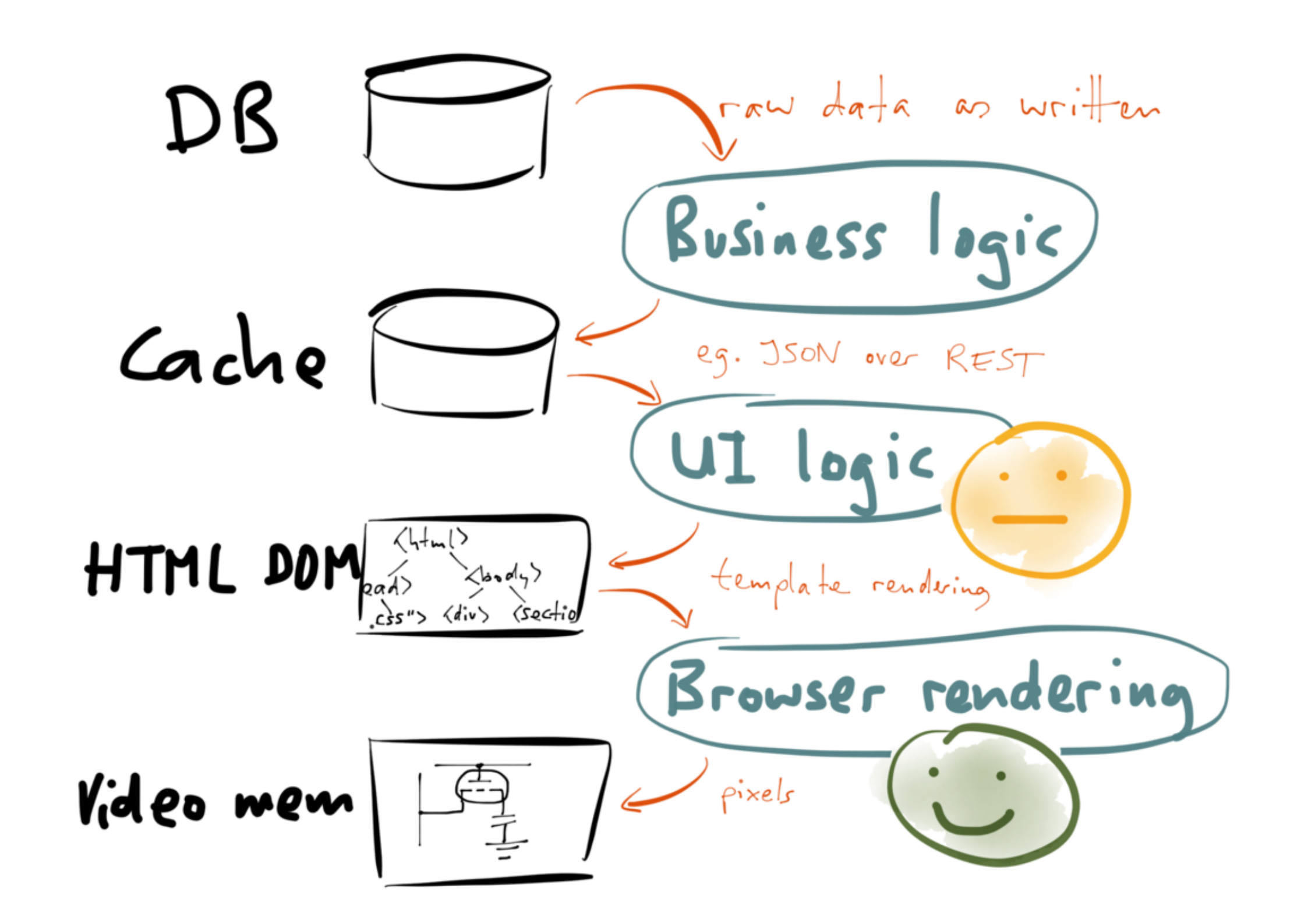

DB
raw data as written
Business logic
Cache

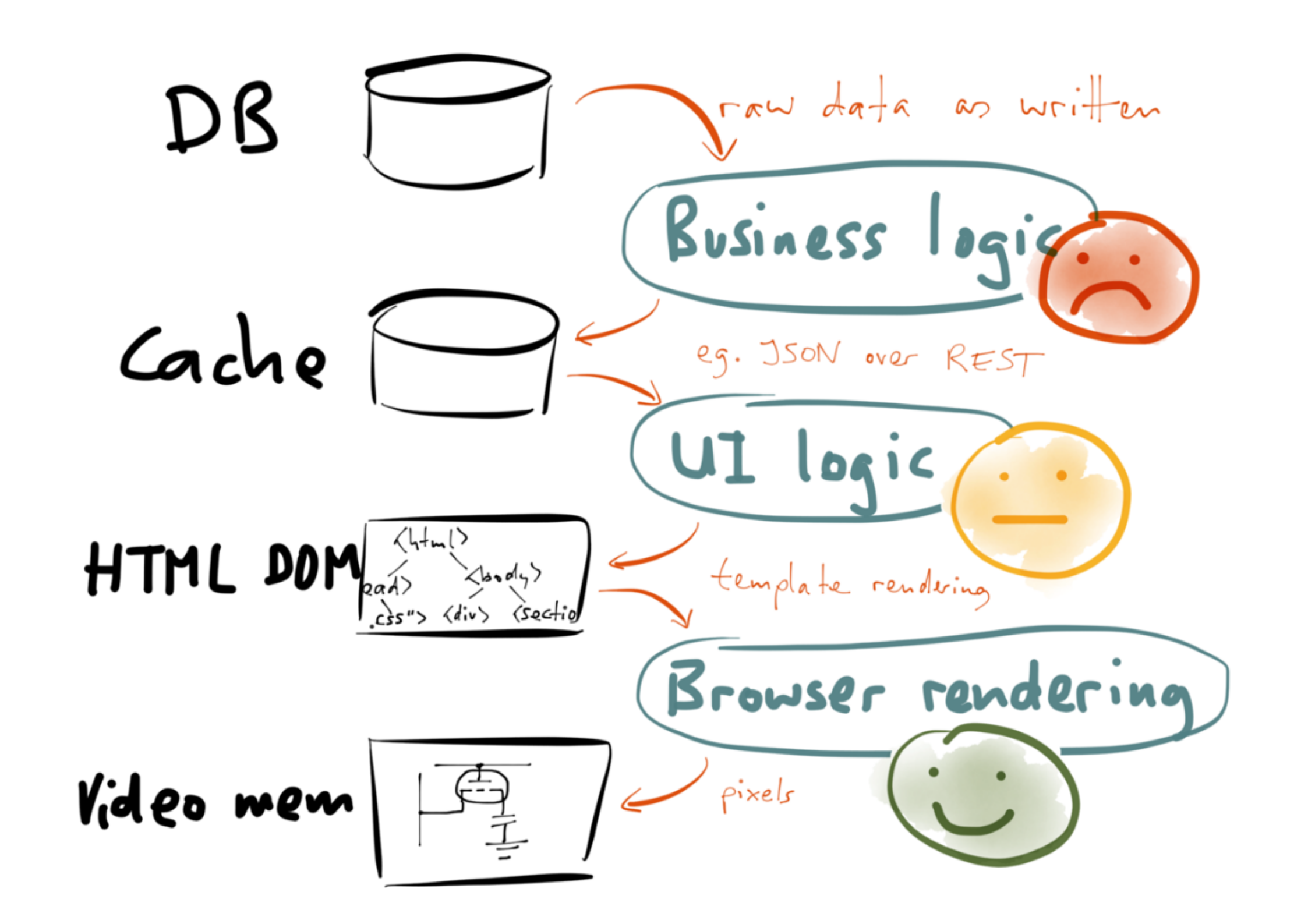

DB
raw data as written
Business logic
Cache
eg. JSON over REST
UI logic
HTML DOM
<html>
<body>
template rendering

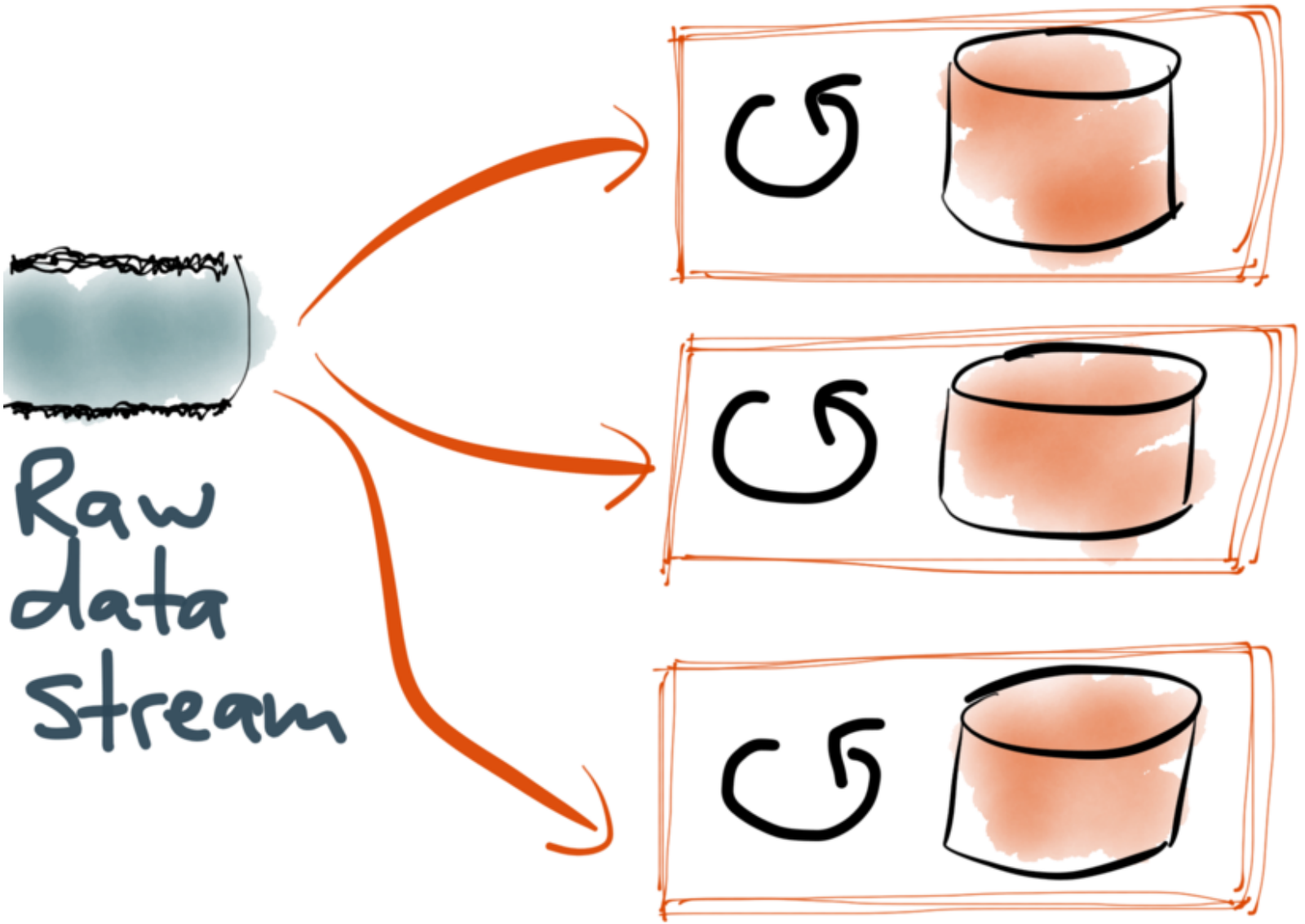

DB
raw data as written
Business logic
Cache
eg. JSON over REST
UI logic
HTML DOM
<html>
<body>
template rendering
Browser rendering
Video mem
pixels


DB
raw data as written
Business logic
Cache
eg. JSON over REST
UI logic
HTML DOM
<html>
<head>
<body>
.css">

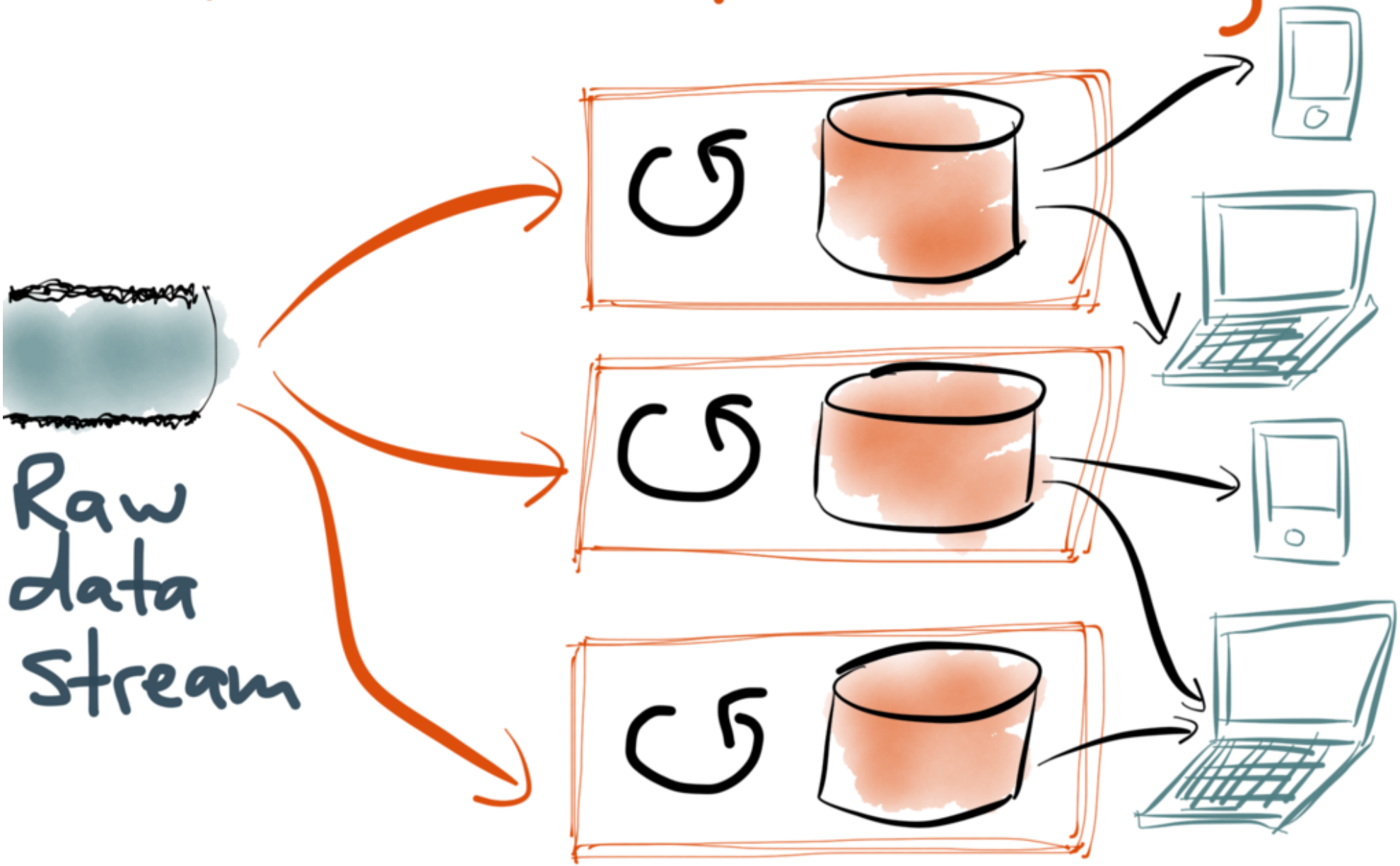
<sectio
template rendering
Browser rendering
Video mem
pixels


DB
raw data as written
Business logic
Cache
eg. JSON over REST
UI logic
HTML DOM
<html>
<body>
template rendering
Browser rendering
Video mem
pixels

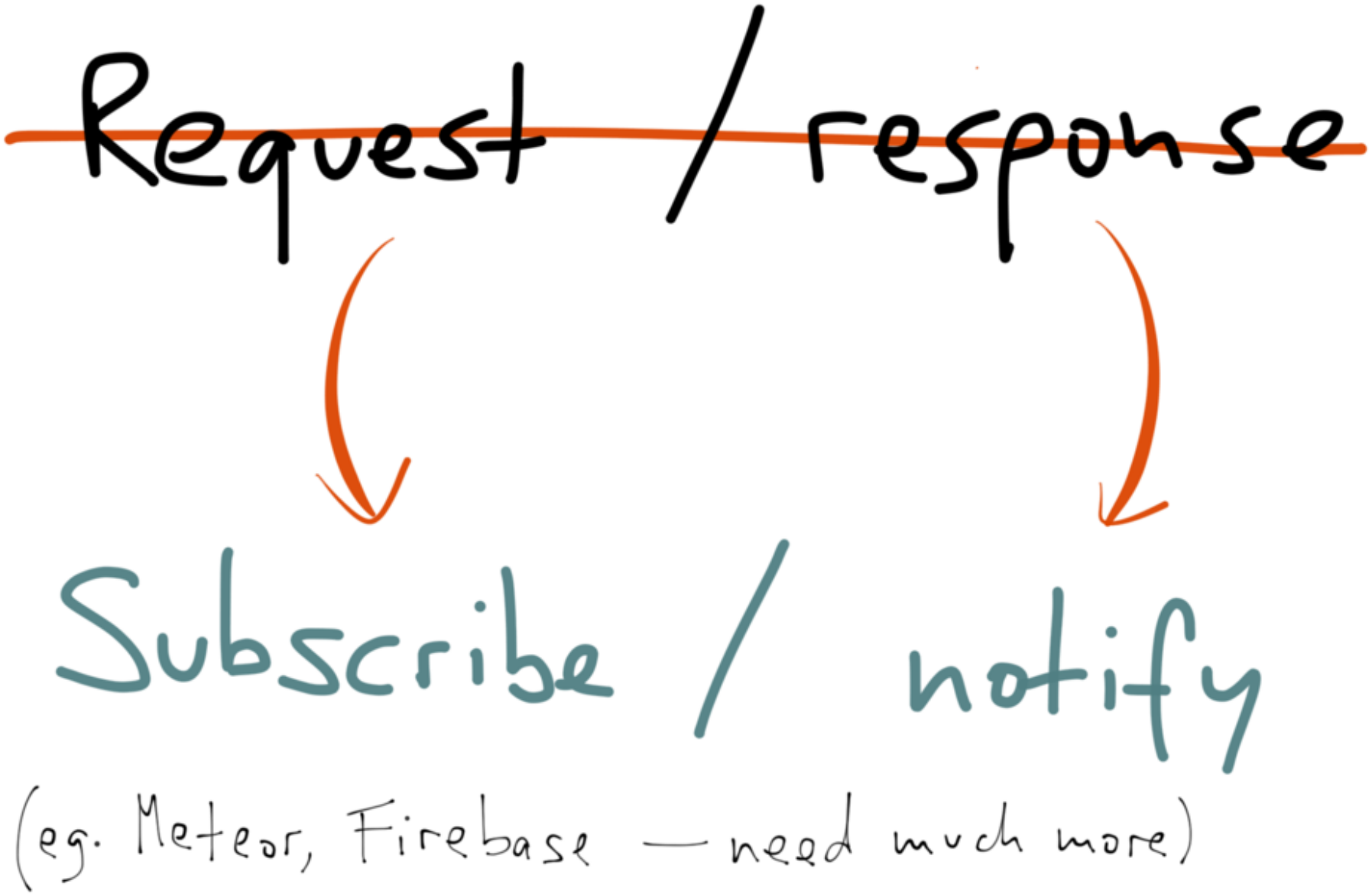

DB
raw data as written
Business logic
Cache
eg. JSON over REST
UI logic
HTML DOM
<html>
<body>
template rendering
Browser rendering
Video mem
pixels


Raw
data
stream

Clients subscribe to MV changes



Request / response

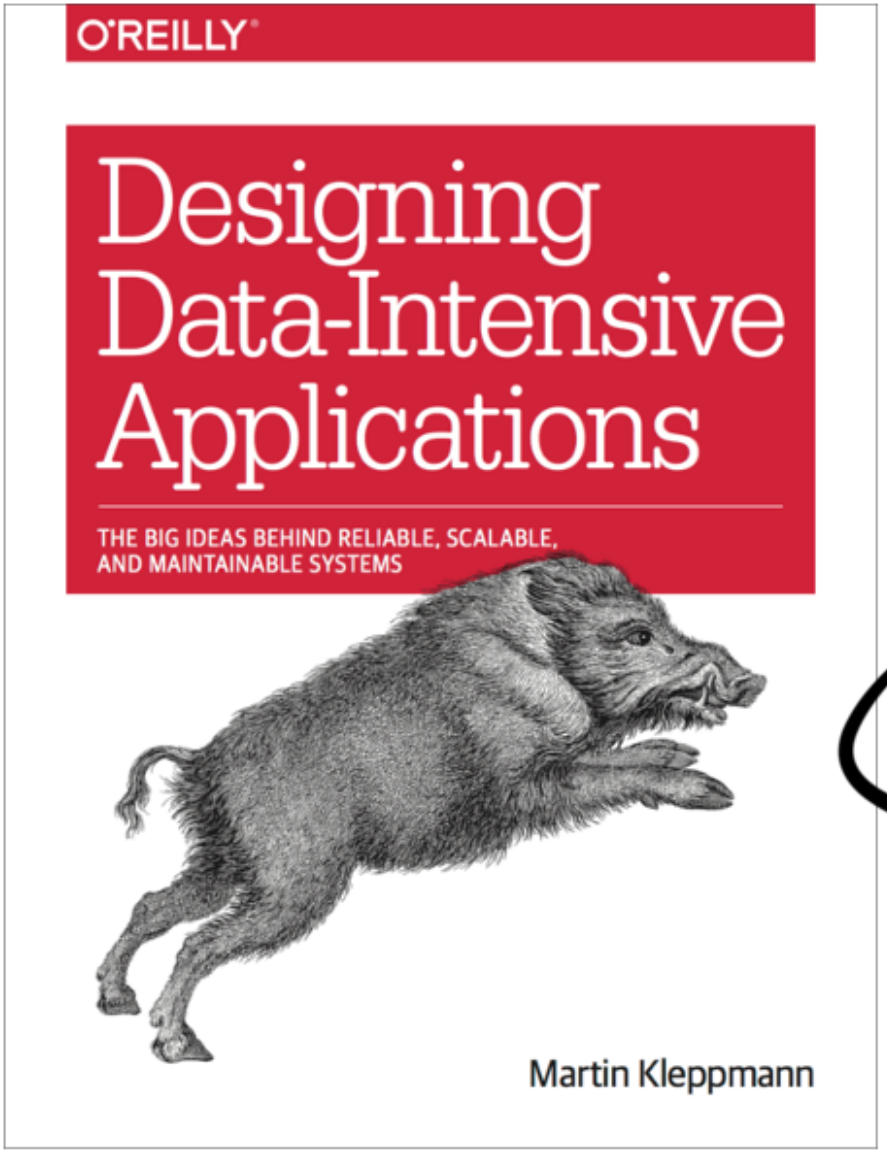

Request / response
Subscribe / notify
(eg. Meteor, Firebase — need much more)

Including updates to materialized views.

References / further reading

- Martin Kleppmann: "Rethinking caching in web apps." 1 October 2012. http://martin.kleppmann.com/2012/10/01/rethinking-caching-in-web-apps.html
- Martin Kleppmann: "Designing data-intensive applications." O'Reilly, to appear in 2015. http://dataintensive.net/
- Jay Kreps: "The Log: What every software engineer should know about real-time data's unifying abstraction." 16 December 2013. http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
- Jay Kreps: "Questioning the Lambda Architecture." 2 July 2014. http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html
- Jay Kreps: "Why local state is a fundamental primitive in stream processing." 31 July 2014. http://radar.oreilly.com/2014/07/why-local-state-is-a-fundamental-primitive-in-stream-processing.html
- Nathan Marz and James Warren: "Big Data: Principles and best practices of scalable realtime data systems." Manning MEAP, to appear January 2015. http://manning.com/marz/
- Apache Samza documentation. http://samza.incubator.apache.org/
- Alexandros Labrinidis, Qiong Luo, Jie Xu, and Wenwei Xue: "Caching and Materialization for Web Databases," *Foundations and Trends in Databases*, volume 2, number 3, pages 169–266, March 2010.
- Stefano Ceri, Georg Gottlob, and Letizia Tanca: "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Transactions on Knowledge and Data Engineering*, volume 1, number 1, pages 146–166, March 1989.

dataintensive.net



@martinkl

samza.incubator.apache.org

samza