AWS re:Invent

DECEMBER 2 – 6, 2024 | LAS VEGAS, NV

DAT427-NEW

Deep dive into Amazon Aurora DSQL and its architecture

Marc Brooker
VP/Distinguished Engineer
Amazon Web Services

You don’t need to know this stuff!

Agenda

- Quick reminder: What is Amazon Aurora DSQL?
- Writes and concurrency control
- Reads and SQL execution
- Cross-Region and scalability

Amazon Aurora DSQL is…

A relational SQL database optimized for transactional workloads.

Scalable, up and down.

Serverless.

Active-active, and multi-Region.

PostgreSQL compatible.

Built on our experience.

Rethinking transactional databases

```
BEGIN;
INSERT INTO dogs VALUES ('snuffles', 4);
INSERT INTO dogs VALUES ('sophie', 8);
COMMIT;
```

Atomic

Consistent

Isolated

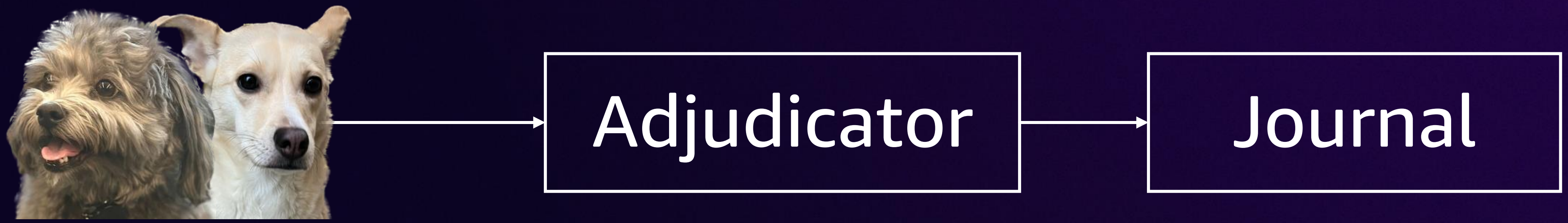
Durable

The log is the database

Atomic and Durable

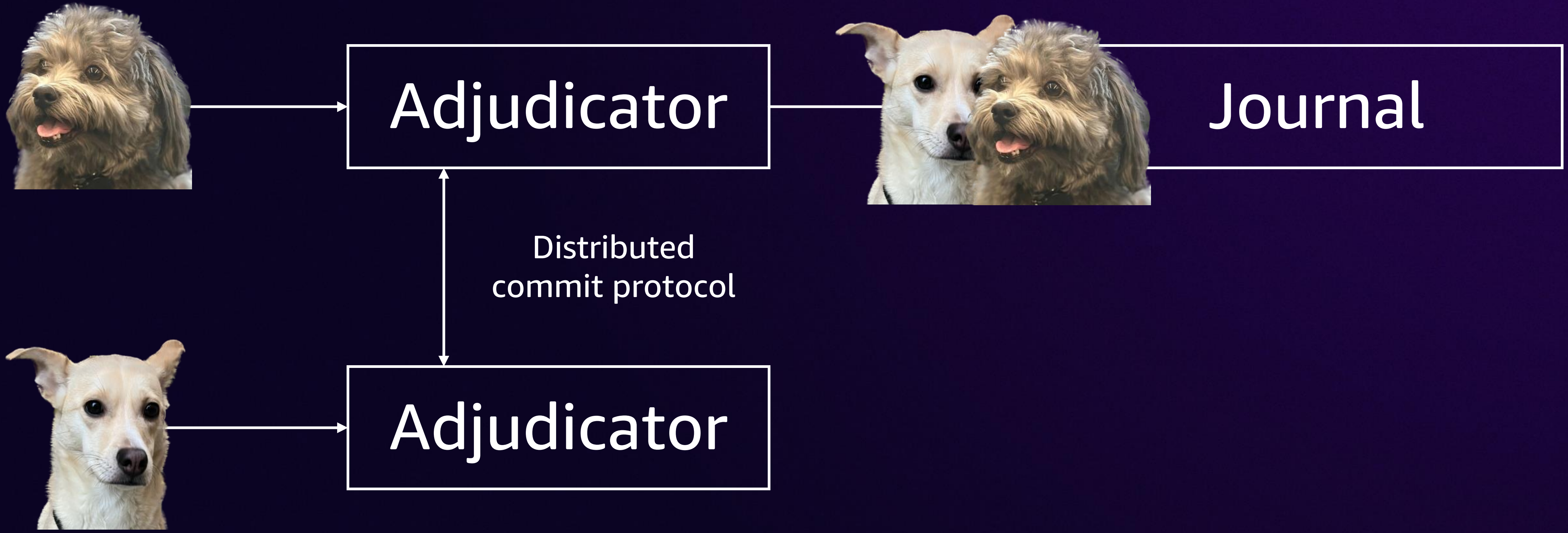
Awesome internal primitive
providing atomicity and durability.

Atomic and Durable and Isolated



Looks for conflicts between this transaction and other recent transactions.

Atomic and Durable and Isolated and Scalable



```
BEGIN;
UPDATE dogs SET age = age + 1
  WHERE name IN (‘snuffles’, ‘max’);
COMMIT;
```

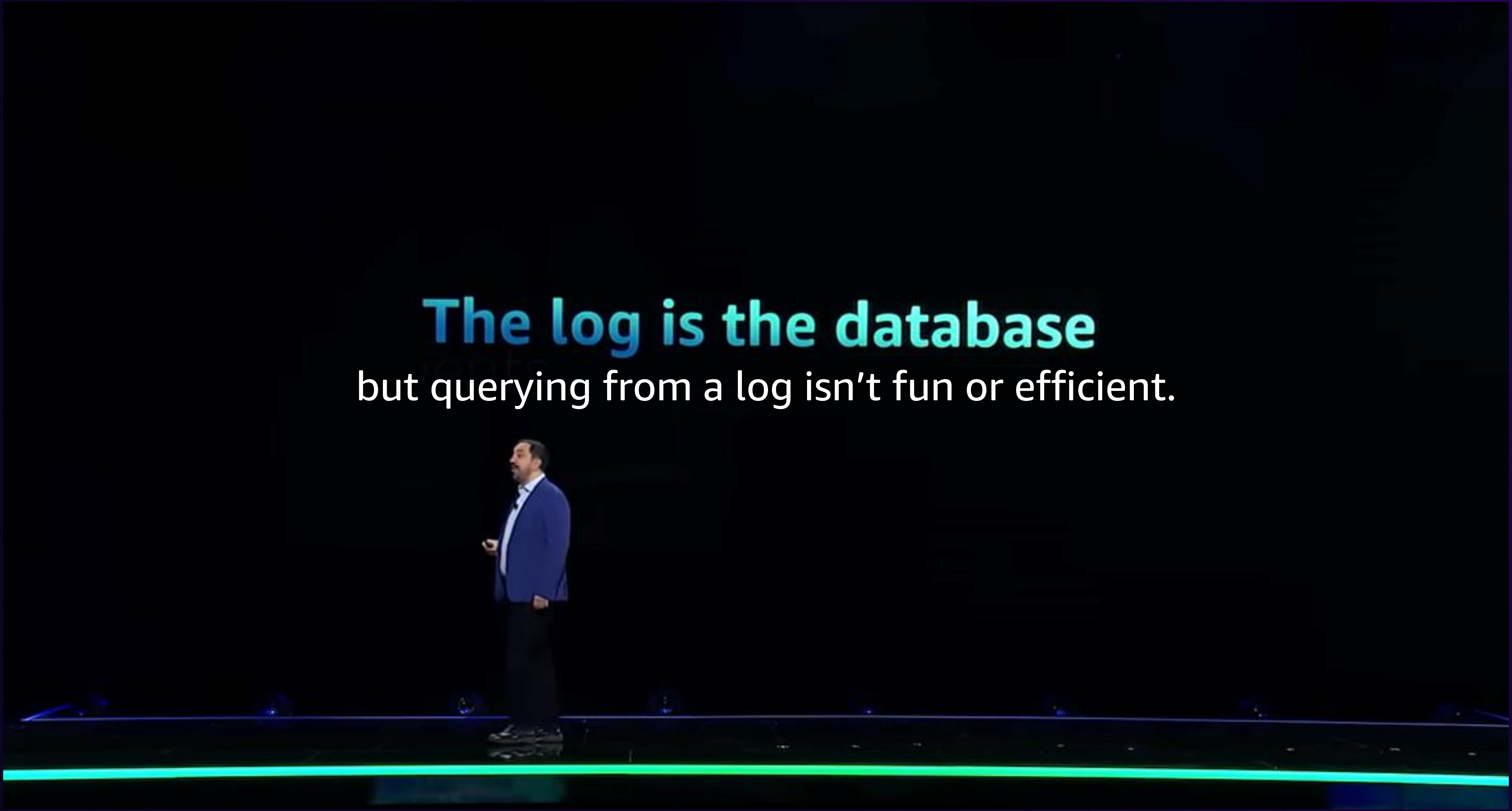
BEGIN;

Find the current age of max and snuffles.

Add one.

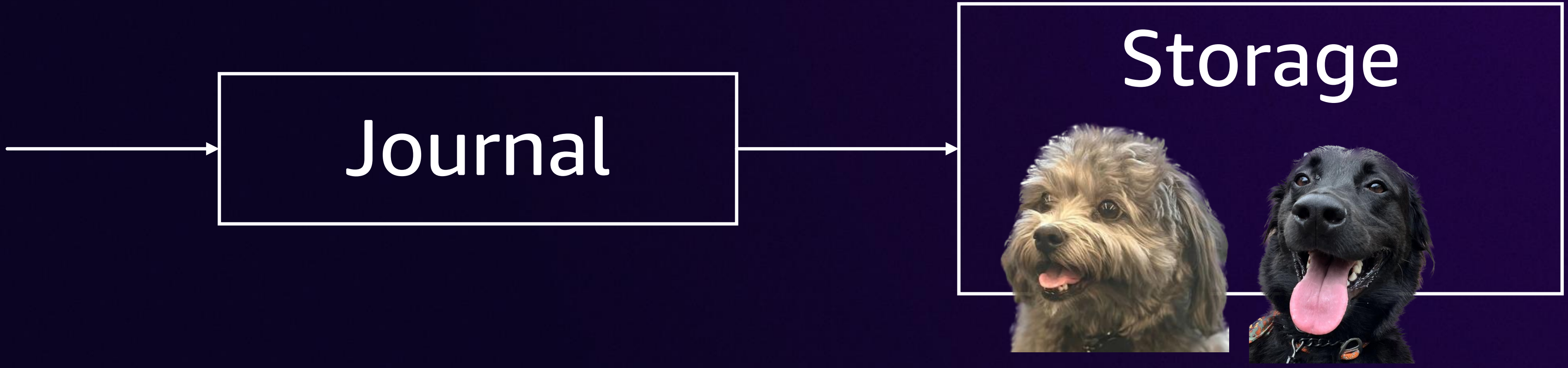
Overwrite the old values with the new ones.

COMMIT;



but querying from a log isn’t fun or efficient.

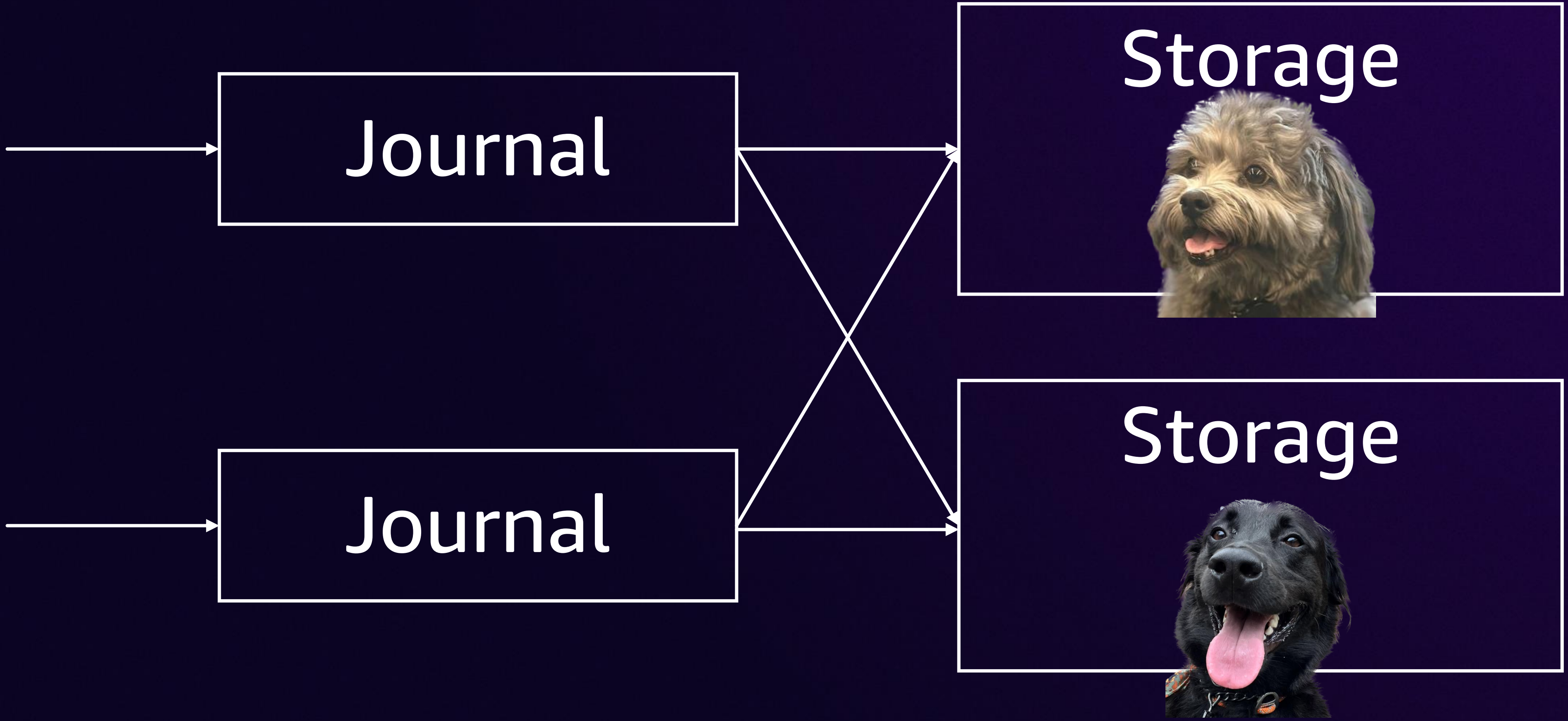
Queryable



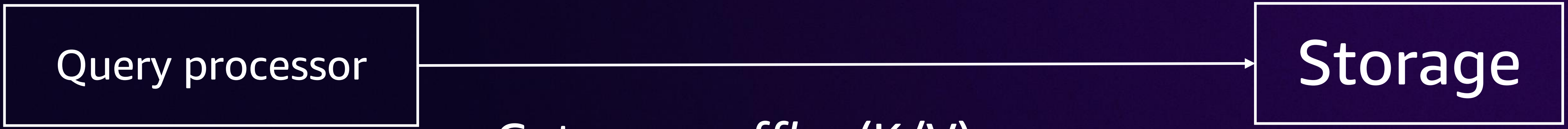
Storage provides efficient ways to query data.

But is not responsible for durability or concurrency control.

Queryable and Scalable



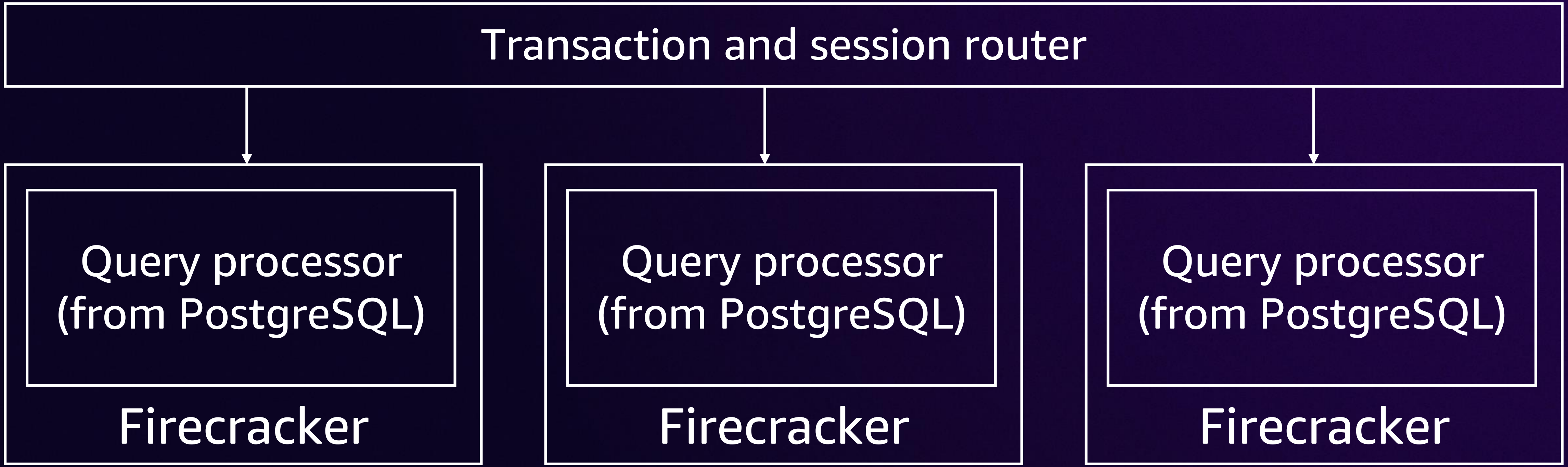
Queryable: Pushdown



- Get me *snuffles* (K/V)
- Get me all good dogs (scan)
- Count all bad dogs (aggregate)
- Get me their ages and favorite treat (project)
- etc.

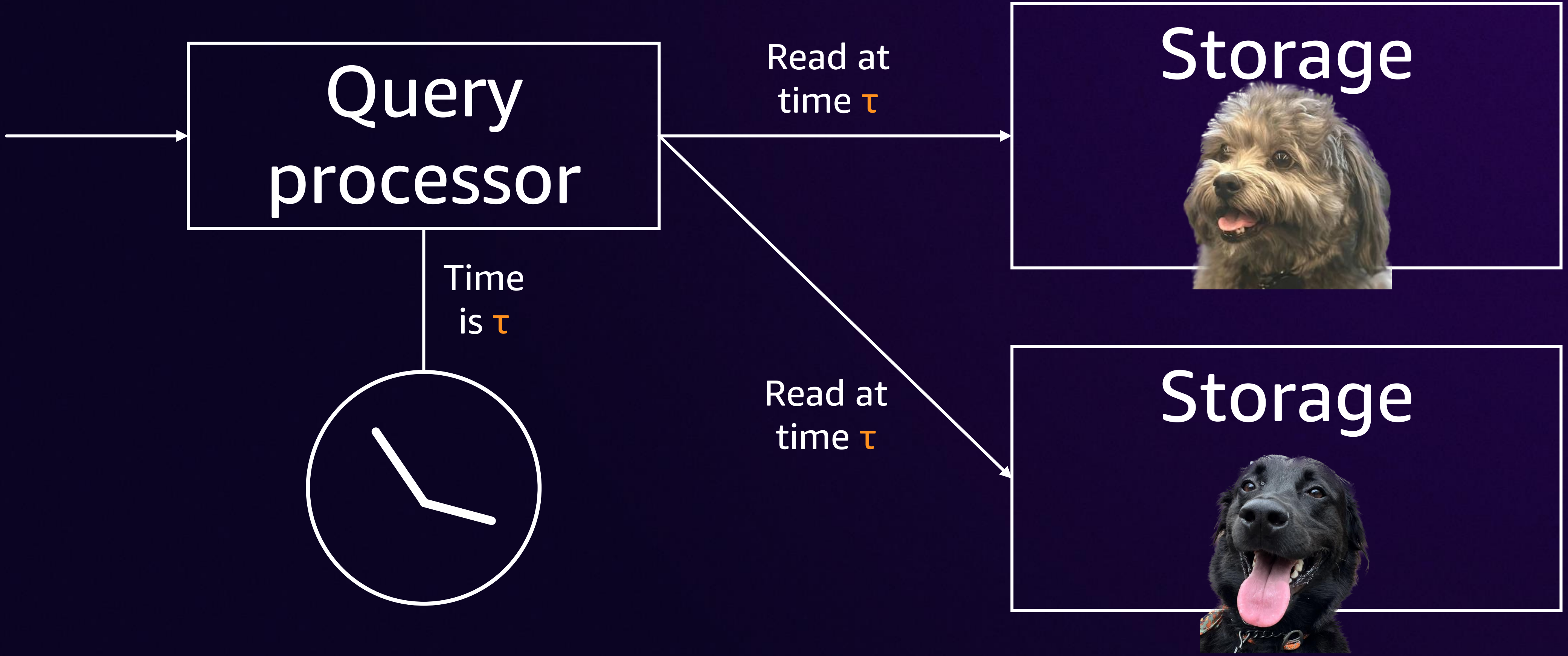
Saves a lot of round trips!

```
BEGIN;
SELECT count(1) FROM dogs
      WHERE state = ‘hungry’;
UPDATE food SET quantity = quantity - 2
      WHERE type = 3;
UPDATE dogs SET state = ‘well fed’
      WHERE name IN (‘fido’, ‘max’);
COMMIT;
```

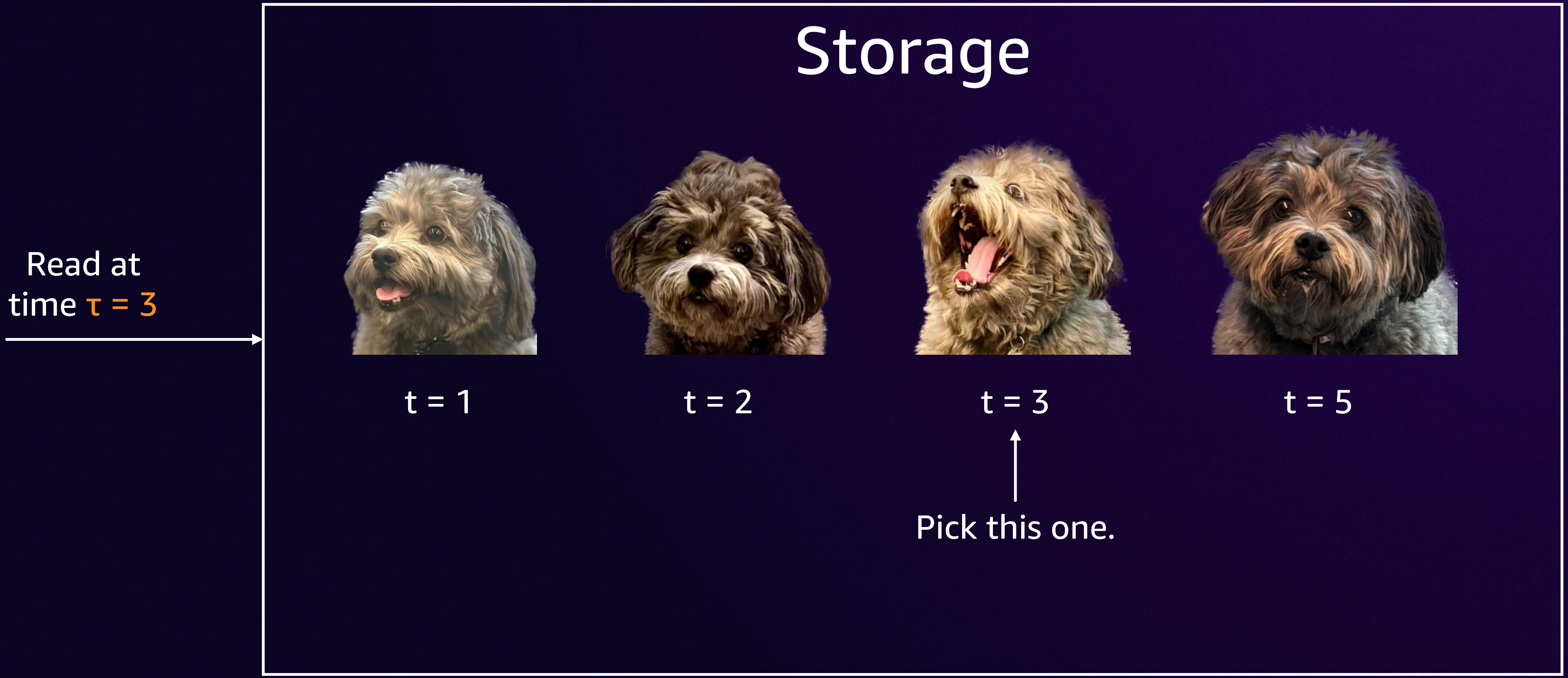

Transaction and session router
Query processor (from PostgreSQL)
Firecracker
Query processor (from PostgreSQL)
Firecracker
Query processor (from PostgreSQL)
Firecracker

Each database can have any number of these. Just keep scaling.

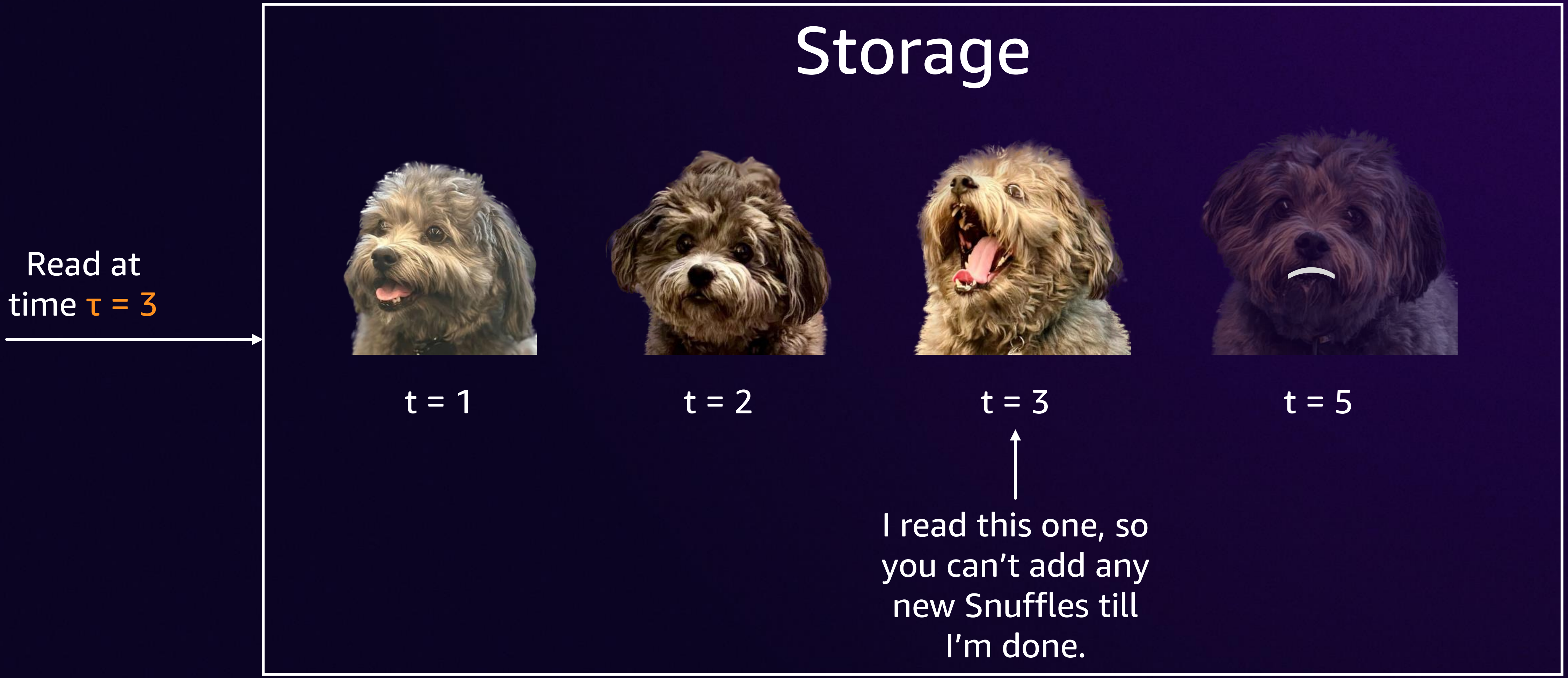
Isolation of reads

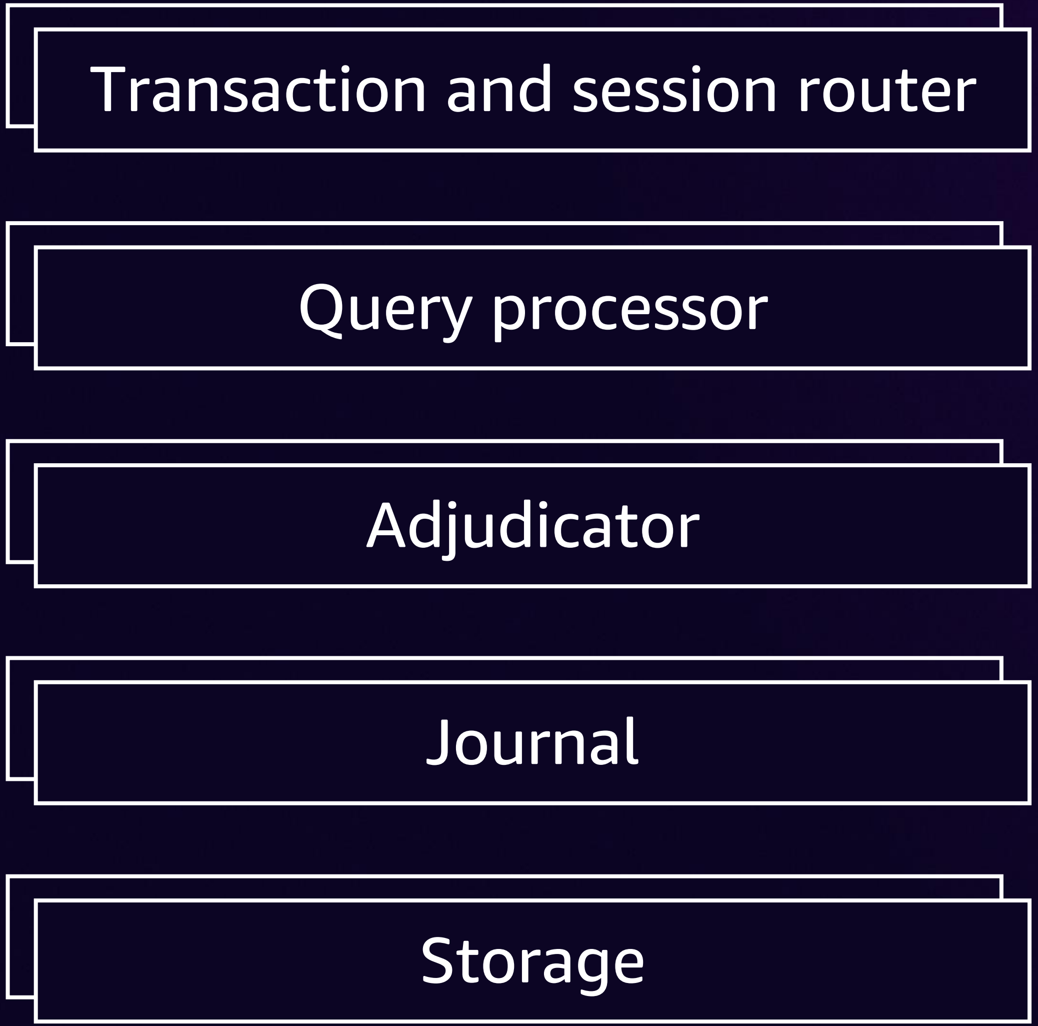


Isolation of reads: Multiversioning



Isolation of reads: The locking alternative

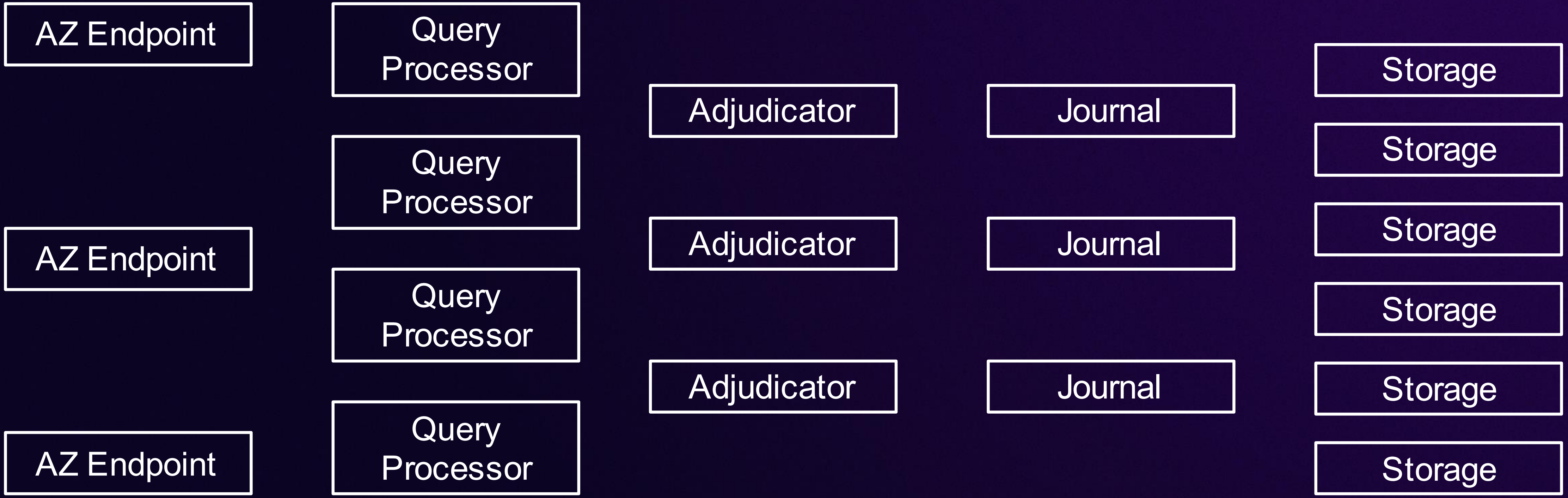

Storage
Read at time $\tau = 3$
$t = 1$
$t = 2$
$t = 3$
$t = 5$
I read this one, so you can't add any new Snuffles till I'm done.



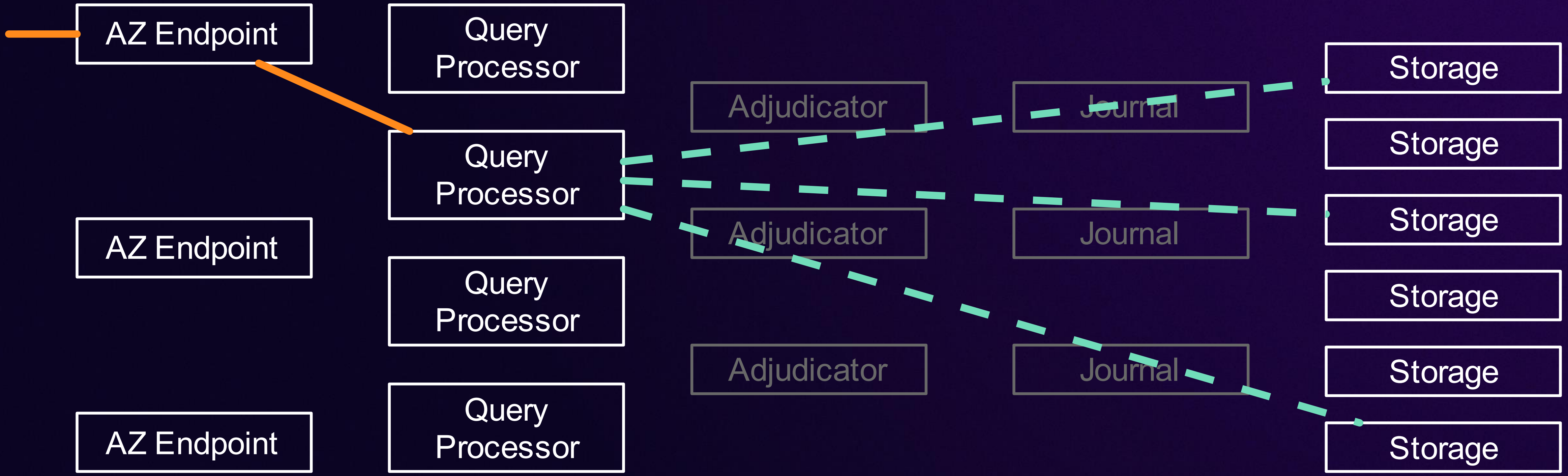
Each layer scales:

- Horizontally
- Independently
- Dynamically

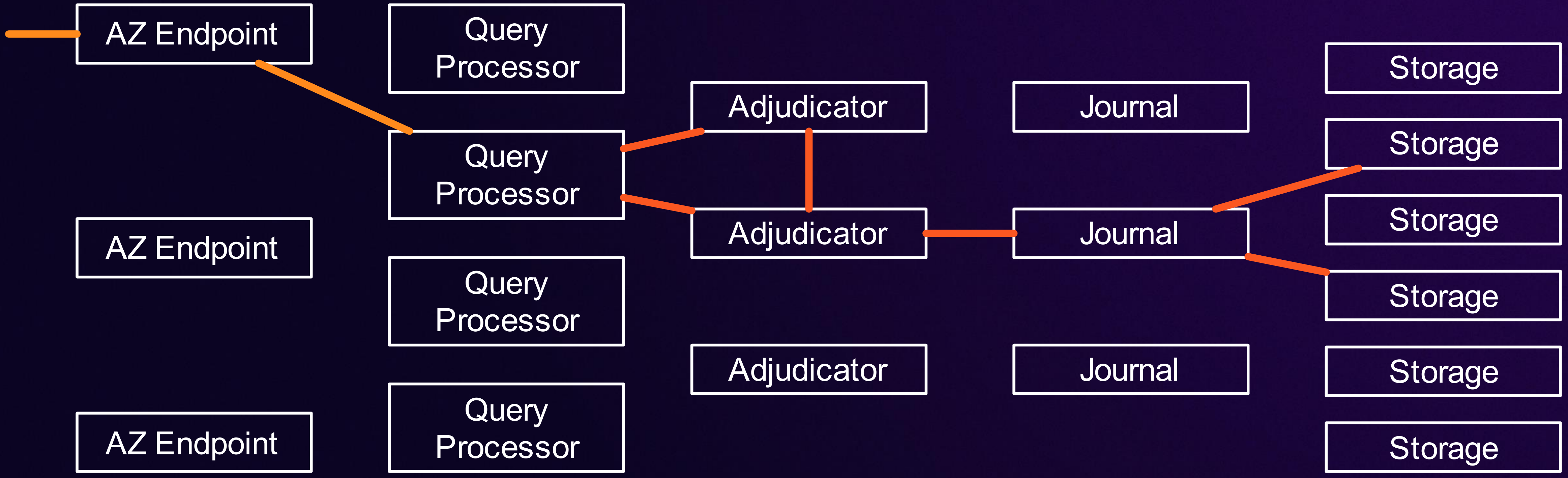
Based on the demands of your workload.



— Reads and Writes – – Reads — Commits



▬	Reads and Writes
╌	Reads
▬	Commits


AZ Endpoint
AZ Endpoint
AZ Endpoint
Query Processor
Query Processor
Query Processor
Query Processor
Adjudicator
Adjudicator
Adjudicator
Journal
Journal
Journal
Storage
Storage
Storage
Storage
Storage
Storage

Reads and Writes
Reads
Commits

Deep dive: Isolation

BEGIN;	Query processor chooses τ_{start}
SELECT …	Reads done at τ_{start}
INSERT …	Writes spooled in QP
UPDATE …	Reads at τ_{start}, writes in QP
COMMIT;	Check isolation rules.

No coordination needed before COMMIT!

Optimistic Concurrency Control (aka OCC)

No locks, no coordination before commit.

Strong Snapshot Isolation

(equivalent to PostgreSQL’s REPEATABLE READ level).

What is Snapshot Isolation?

- Never see uncommitted data.
- Reads are repeatable.
- Reads all come from a single point in (logical) time.
- Conflicting writes are rejected (writes are not lost).

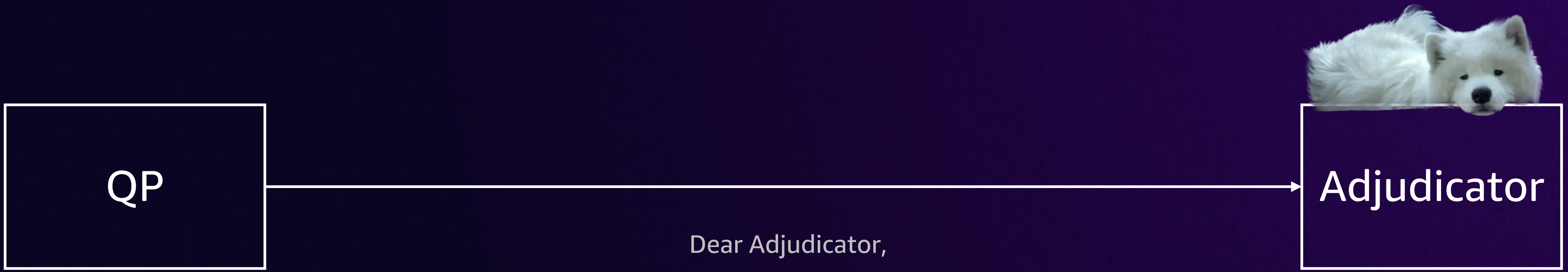
But it’s not serializable.

```
START TRANSACTION;

SELECT n FROM t WHERE id IN (1, 2);
UPDATE t SET n = 2 WHERE id = 1;



COMMIT; ✅
```

```
START TRANSACTION;

SELECT n FROM t WHERE id IN (1, 2);
UPDATE t SET n = 2 WHERE id = 2;



COMMIT; ✅
```

Snapshot Isolation recipe

- Perform all reads at τ_{start}
- At commit time, choose τ_{commit}
- The transaction can commit if (and only if) no other transaction has written to the same keys between τ_{start} and τ_{commit}
- Perform the writes at τ_{commit}



Dear Adjudicator,

Here are the keys I intend to write, and my τ_{start}

If no other transaction has written these keys since τ_{start}, pick a τ_{commit} and write these changes to the Journal.

Never allow another conflicting transaction to pick a lower τ_{commit}.

Your friend, the QP.

Snapshot Isolation is a sweet spot.

Deep dive: Cross-Region

Optimize for round trips.

Data travels at 200km per ms,
or 123 miles per ms.

Can happen entirely locally!

BEGIN;	Query processor chooses τ_{start}
SELECT …	Reads done at τ_{start}
INSERT …	Writes spooled in QP
UPDATE …	Reads at τ_{start}, writes in QP
COMMIT;	Check isolation rules.
	Make data durable.

Needs cross-Region coordination.

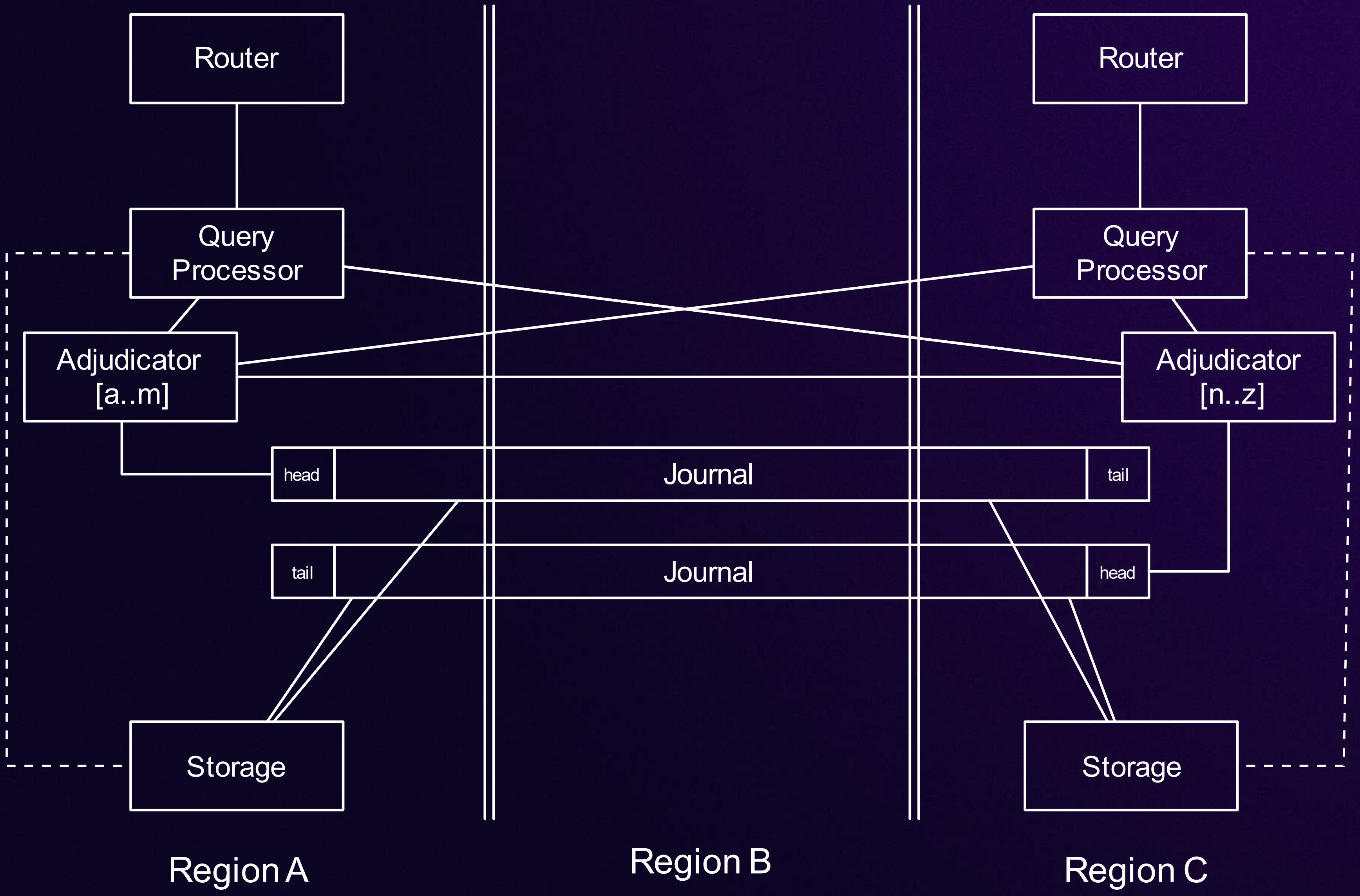
Coordinate once, at commit time.

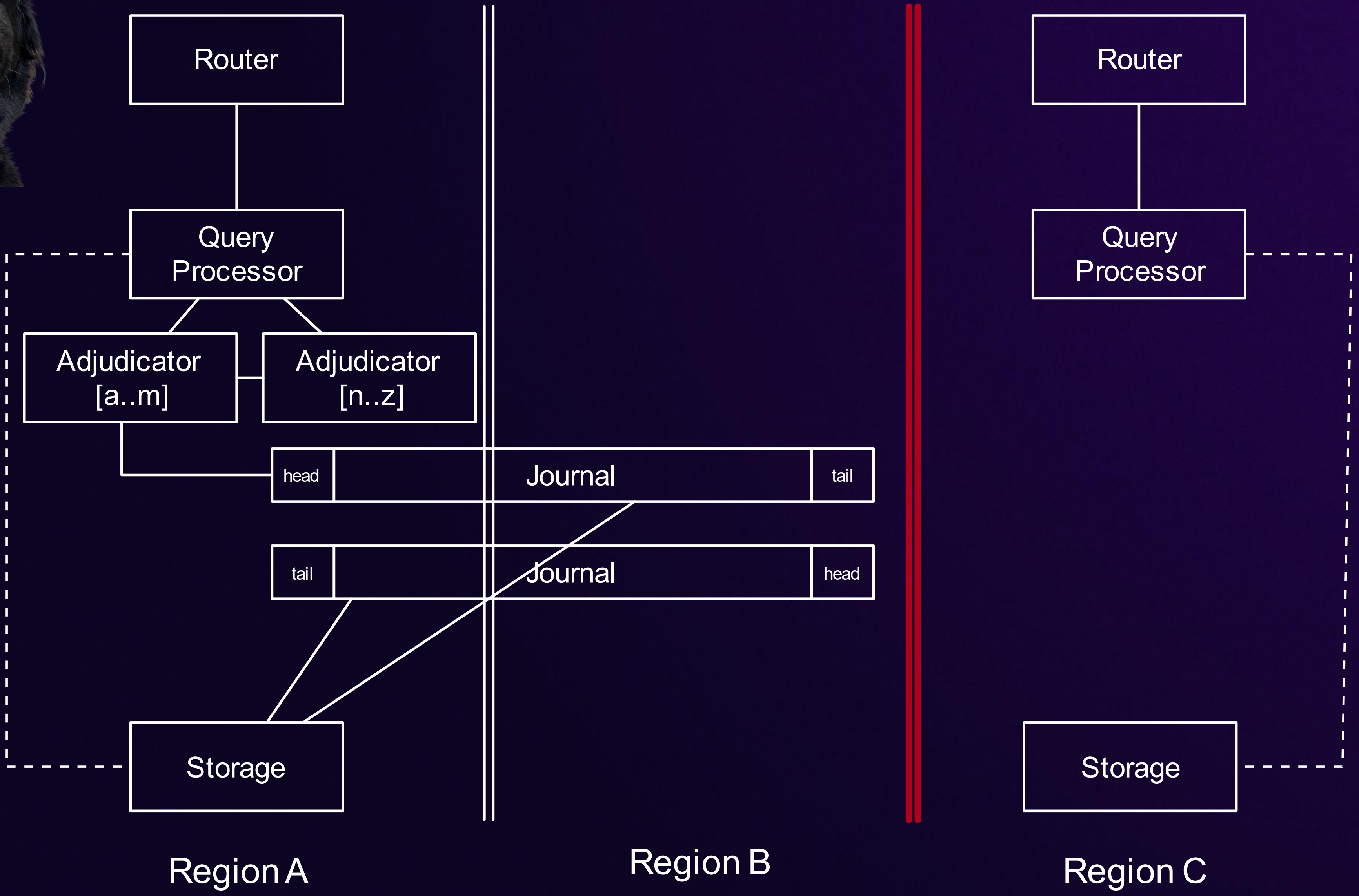
Coordinate once, at commit time.

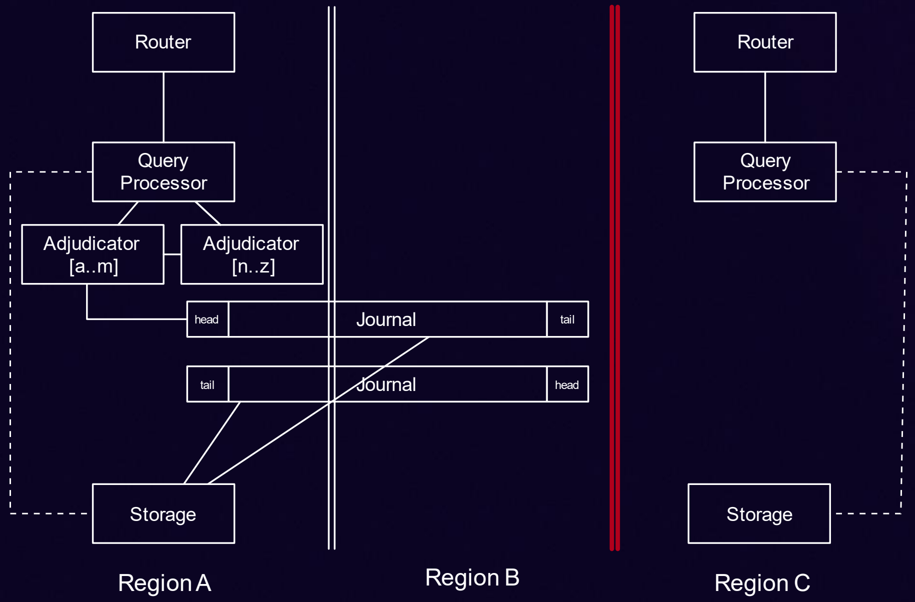
Read-only transactions don’t need to coordinate at all!

Optimize for fast failover.

Changing leadership doesn’t require moving data (or lock state).


Router
Query Processor
Adjudicator [a..m]
Router
Query Processor
Adjudicator [n..z]
head
Journal
tail
tail
Journal
head
Storage
Storage
Region A
Region B
Region C


Router
Query Processor
Adjudicator [a..m]
Adjudicator [n..z]
head
Journal
tail
tail
Journal
head
Storage
Region A
Region B
Router
Query Processor
Storage
Region C



- Read path: no change
- Adjudicators:
  move into healthy regions
- Journals:
  majority already in healthy regions
- No data loss, no availability loss

Implementation quality

Rust.

Deterministic simulation testing.

Fuzzing.

Formal methods.

Runtime monitoring.

Thank you!

Please complete the session survey in the mobile app

Marc Brooker

mbrooker@amazon.com

@marcjbrooker