IS KAFKA A DATABASE?

A 2 MINUTE STREAMING POST

Myth: Kafka is NOT a database

Let's disprove that 👉

What makes a database?

Some people will immediately say that Kafka doesn't have:

- SQL
- tables
- data models / schemas
- queries
- secondary indices
- foreign keys
- backups

and therefore cannot be a database.

But is that what makes a database?

What makes a database?

No.

It's the properties that the software has in relation to storing and querying data.

A database is literally a program for:
1. writing data to
2. reading data from

...

But does that make my Python dictionary CLI program a database?

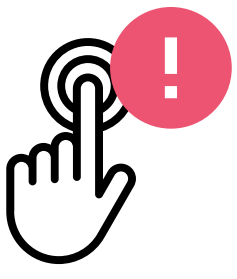
What makes a database?

A program that you write/read data from can offer **different guarantees** as to what happens when:

Two users **write** the same data at the same time while we have a uniqueness constraint.

The machine the database runs on **dies** and **loses its disk**.

Your application write 10 records and the database **dies in the middle** of the writes while 5/10 are saved.



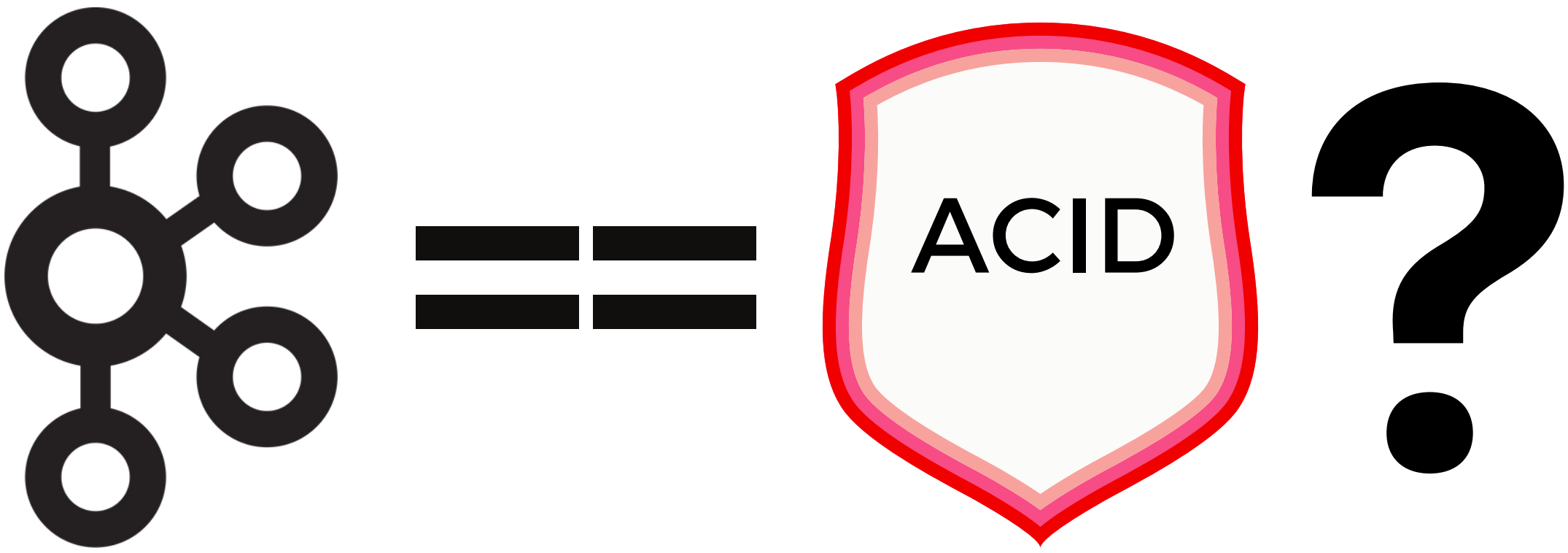
Two users **UPDATE** the same entry at **the same time**.

What makes a database?

The industry has come to define these properties with ACID.

1. **A**tomicity
2. **C**onsistency
3. **I**solation
4. **D**urability

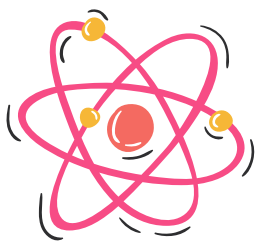
Back to our question about Kafka.



A good **first-principles reasoning** approach would have us see how Kafka fares under these properties and then decide. 👉

Atomicity ✅

The name comes from the smallest particle in existence - **the atom**.



It's basically an **all-or-nothing** guarantee for transactions.

It means that with a set of operations, we want to either:

1. *have **all** be applied*
2. *have **none** be applied*

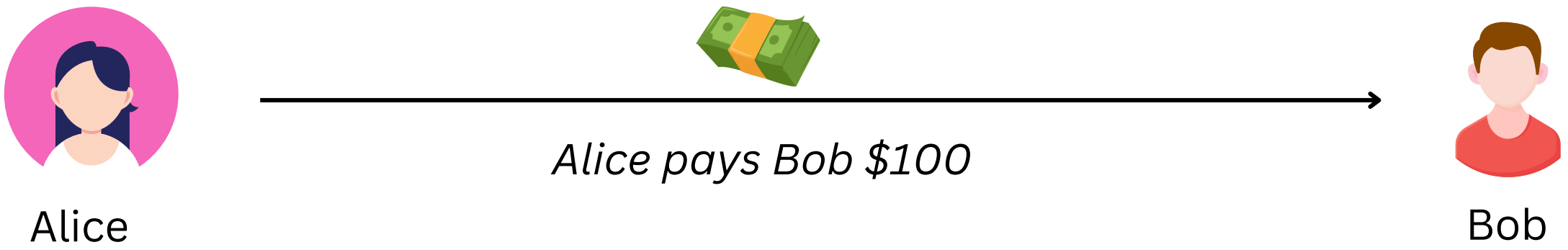
The best example is accounting.

If Alice pays $100 to Bob, you want both of their accounts to be respectively modified. (**all-or-nothing**)
You can't have only one be.

Picture Example 👉

Atomicity ✅

2minutestreaming



The valid expected start and end states are the following:

Start State

Debit: $100

Credit: $0

Debit: $0

Credit: $0

End State

Debit: $0 -$100

Credit: $100 +$100

Debit: $100 +$100

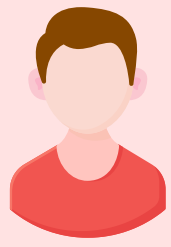
Credit: $0

It would be **completely invalid** if we did it half way:

❌ Invalid State

Debit: $100 ???

Credit: $0 ???



Debit: $100 +$100

Credit: $0

Atomicity ✅

Kafka can achieves atomicity in two ways:

1. **Simplest**: Put all of your data in a single record. Kafka will either save the record or not.

2. **Harder**: Use a Kafka transaction. You can write 100 records to different topics, and as long as it's in a transaction - Kafka will ensure that either all apply or none do.

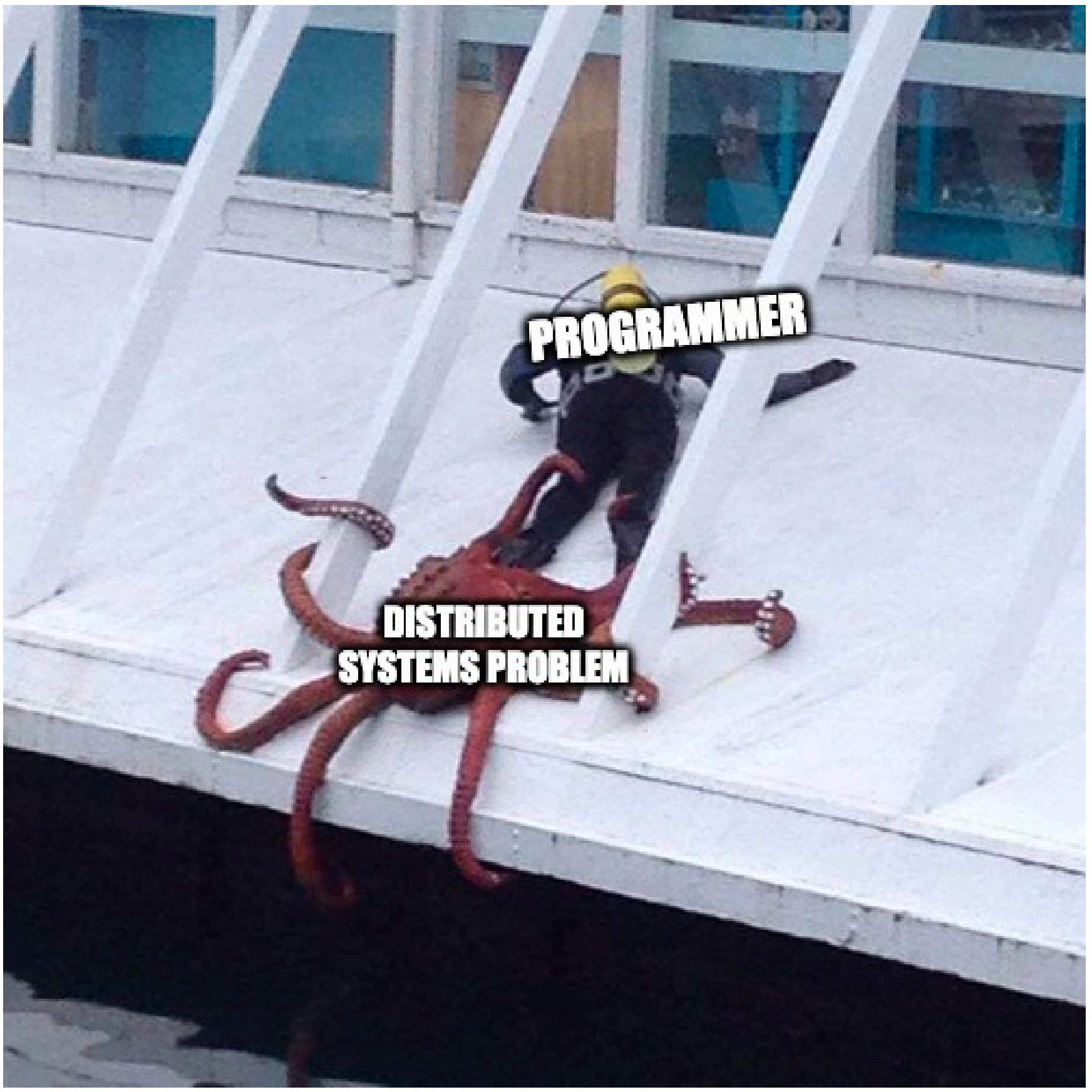
But what about **outside** of Kafka?
What about atomicity in a distributed setting?

👉

Atomicity ✅

Imagine a web app updating a database, cache & search index simultaneously.

How does it ensure that all are updated, or none are? 😰

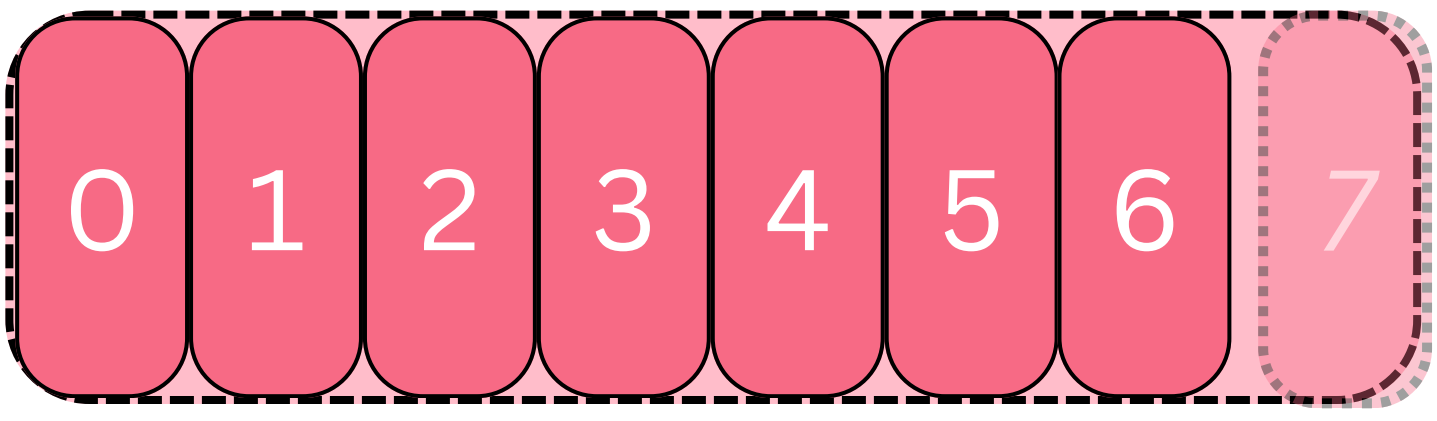


Atomicity ✅

Well, you have to trust the process.

But seriously - if the record(s) is persisted durably in Kafka, then your consumers will **EVENTUALLY** consume it and update the respective systems.

Because the records are read in the same order (assuming the same partition), the downstream systems **will remain consistent** with each other.



The Log

This brings us to: 👉

Consistency ✅

🚨 Before we start, a disclaimer:

Consistency in ACID is **different than** consistency in CAP.

1. **CAP**: C stands for a distributed system's consistency model - how nodes' data is consistent with each other. (out of scope for this post)

2. **ACID**: C confusingly stands for respecting all defined rules. i.e ensuring **a database's correctness**.

Examples: 👉

Consistency ✅

- a database constraint must be respected
- a database trigger must execute on X action
- a cascade delete rule must be executed

But these are invariants defined solely **inside the database layer**.

The consistency (correctness) we care about most is defined in the **application layer**.

The easiest example is again a payments one (C.R.E.A.M.)

Example: 👉

Consistency ✅

1. **Database Correctness**: No account balance should be below zero.

2. **Application Correctness**: every payment should respectively add and subtract the amount from both parties.

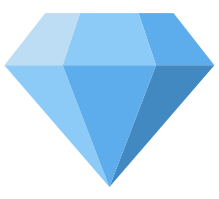
1) can be a correctness invariant that we have unique keys. In Kafka, this would be what **compacted topics** offer.

Consistency is largely the **responsibility of the application** writing to the database.

The **application itself relies** on the *Atomicity*, *Durability* & *Isolation* properties of the database to achieve *Consistency*.

Isolation ✅

Isolation defines how **concurrent changes** to the same record **interact** with each other.



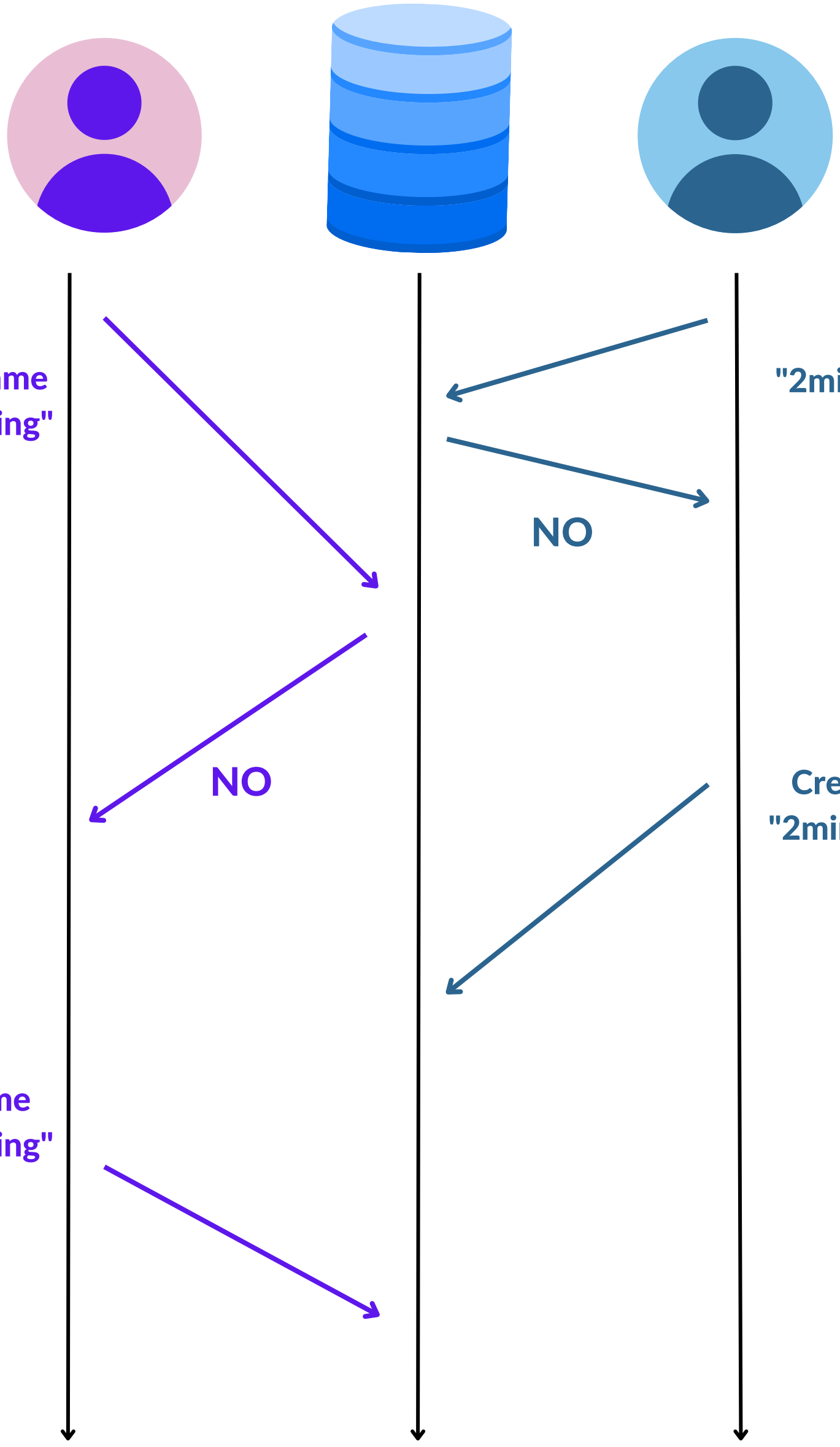
Serializable isolation is the strongest form.

It's when transactions behave **as if** each is executed one by one in **a serial order** (e.g. a single thread).

Even if done in parallel.

Here's a simple example of what a lack of serializability can cause: 👉

Isolation ✅



More info: 👉

2minutestreaming

Isolation ✅

In the previous example (feel free to slide back), we have two processes that try to register the **same username** at the **same time**.

1. check if the username exists
2. register it if it doesn't exist

In last slide's example, we hit the race condition where both processes execute 1) at the same time, hence both receive a **false** result - the username does not exist yet.

They both then decide that they will execute 2), and they do execute 2). If there isn't a database-level constraint, both writes will pass and we will have **two duplicate usernames**.

2minutestreaming

Isolation ✅

How would you guard agains this in Kafka?

Simple.

Serialize the checks.

Kafka has two fundamental properties:

- 1. Partitions are ***ordered***.
- 2. a partition is **exclusively read by one** consumer in a consumer group.

This means that a single consumer will be reading our records (per partition).

Since the processing logic is **single threaded**, we achieve **serializable isolation**. 💎

2minutestreaming

Durability ✅

The easiest one for last.

Durability - a guarantee that changes made to the database are successfully committed to survive permanently, even in the case of system failures.

Kafka aces this.

Kafka **persists messages to disk** and replicates them across brokers (replication factor)

Producers configured with *acks=all* only consider writes successful when **fully replicated**.

Consumers only read the fully-replicated messages (up to the **high watermark offset**)

✅ It's durable!

Conclusion

So...

Kafka **is a** database 😎

But not a **traditional database**.

- Its programming model is **lower level** than most traditional databases.
- It is also **more complicated** than what you would expect from your typical SQL.

Nevertheless, it **has the properties required** to be a database.

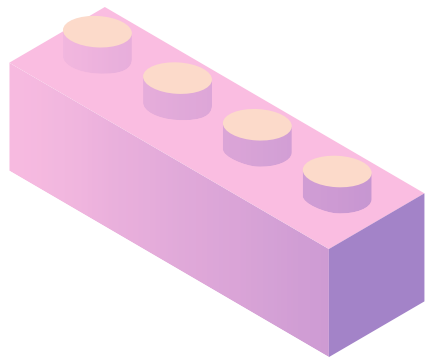
It can be thought of as a **b2b** (business-to-business) database, not a **b2c** (business-to-consumer).

What do I mean by that? 👉

2minutestreaming

Conclusion

Kafka provides **the building blocks** for a new traditional database via transaction processing using streams and stream processors



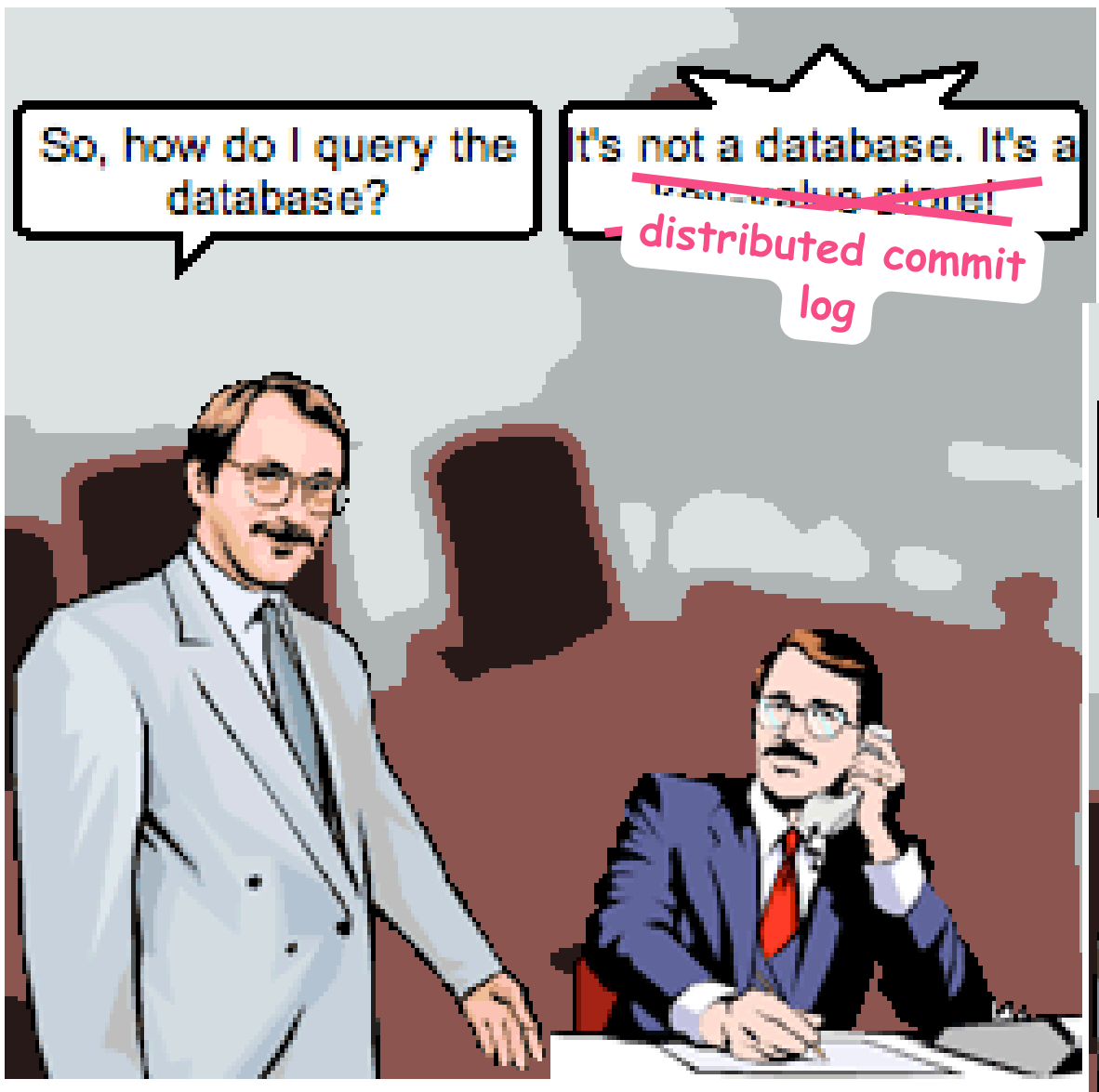
A transaction can be broken down into a multi-stage stream pipeline.

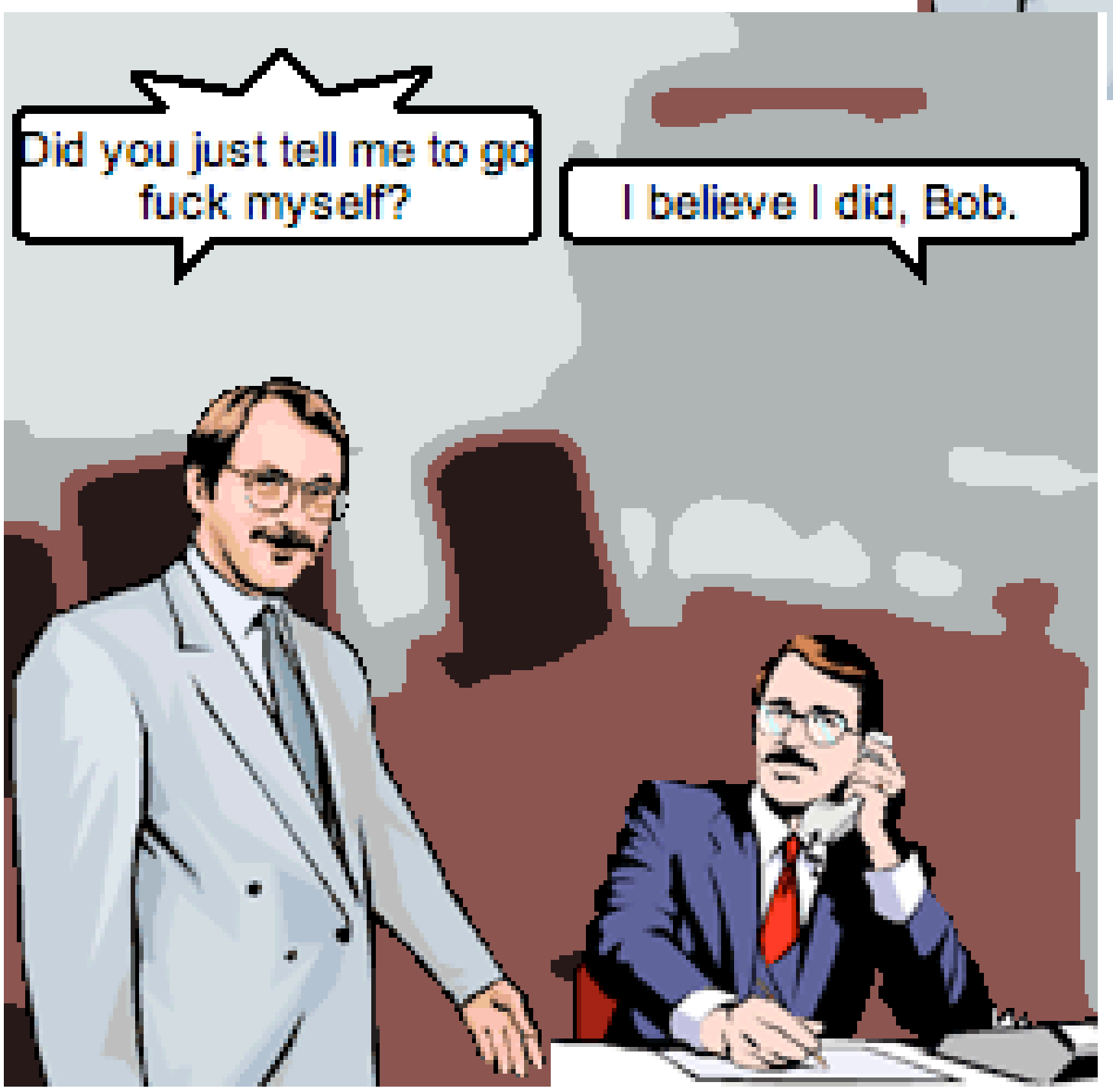
It can:

- can have strong consistency (the CAP type)
- be reliable
- have high throughputg

But enough for today, let's end with a meme 👉

KAFKA AS A DATABASE





2minutestreaming

🎬 Fin.

This was a fun thread exploring definition of databases and their properties.

I hope you enjoyed it!

It took me 10+ hours to make, and I give it out for free.

I only ask for one thing in return:

🙏 **Please like it and repost it** so that this valuable content can reach more people.

If you want more such content, follow me here and on my 1-a-week 2-minute newsletter (link in the post)

2minutestreaming