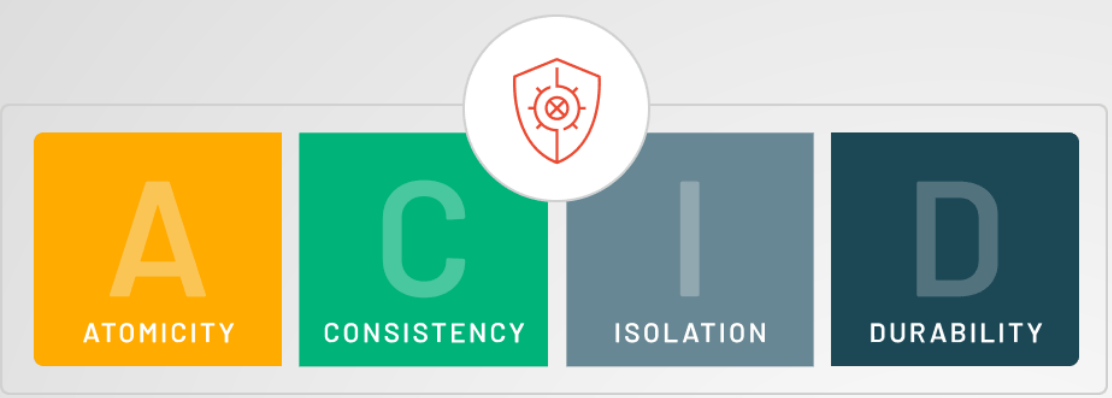
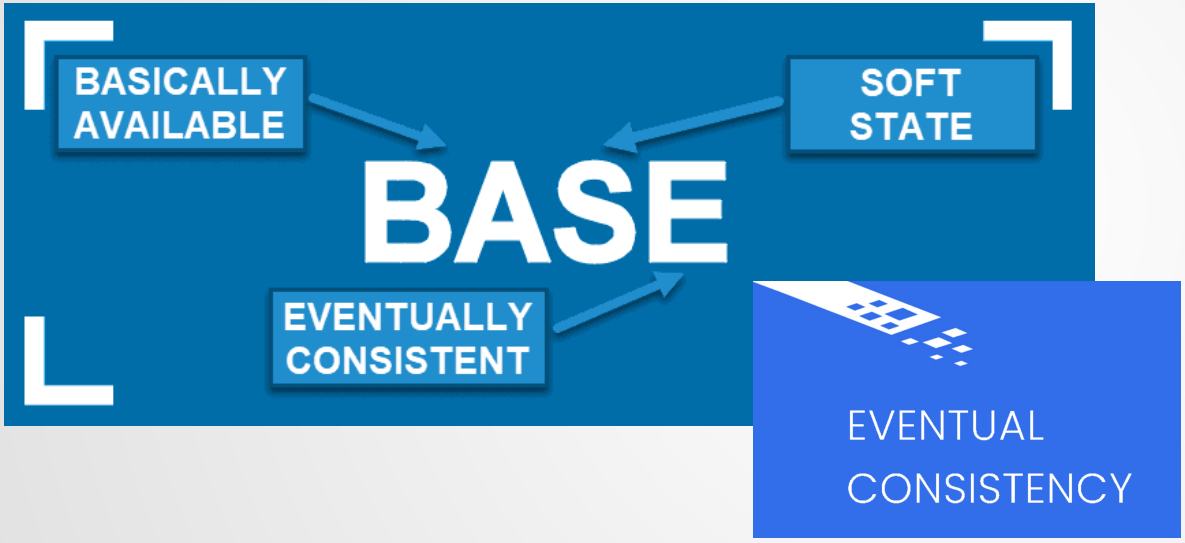

A
ATOMICITY
C
CONSISTENCY
I
ISOLATION
D
DURABILITY


BASICALLY AVAILABLE
SOFT STATE
BASE
EVENTUALLY CONSISTENT

EVENTUAL
CONSISTENCY

Scenario

Multiple Servers

Via replication multiple servers may be running on several 'nodes'.

DB Partitioning

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Vertical Shards

VS1

CUSTOMER ID	FIRST NAME	LAST NAME
1	Alice	Anderson
2	Bob	Best
3	Carrie	Conway
4	David	Doe

VS2

CUSTOMER ID	CITY
1	Austin
2	Boston
3	Chicago
4	Denver

Horizontal Shards

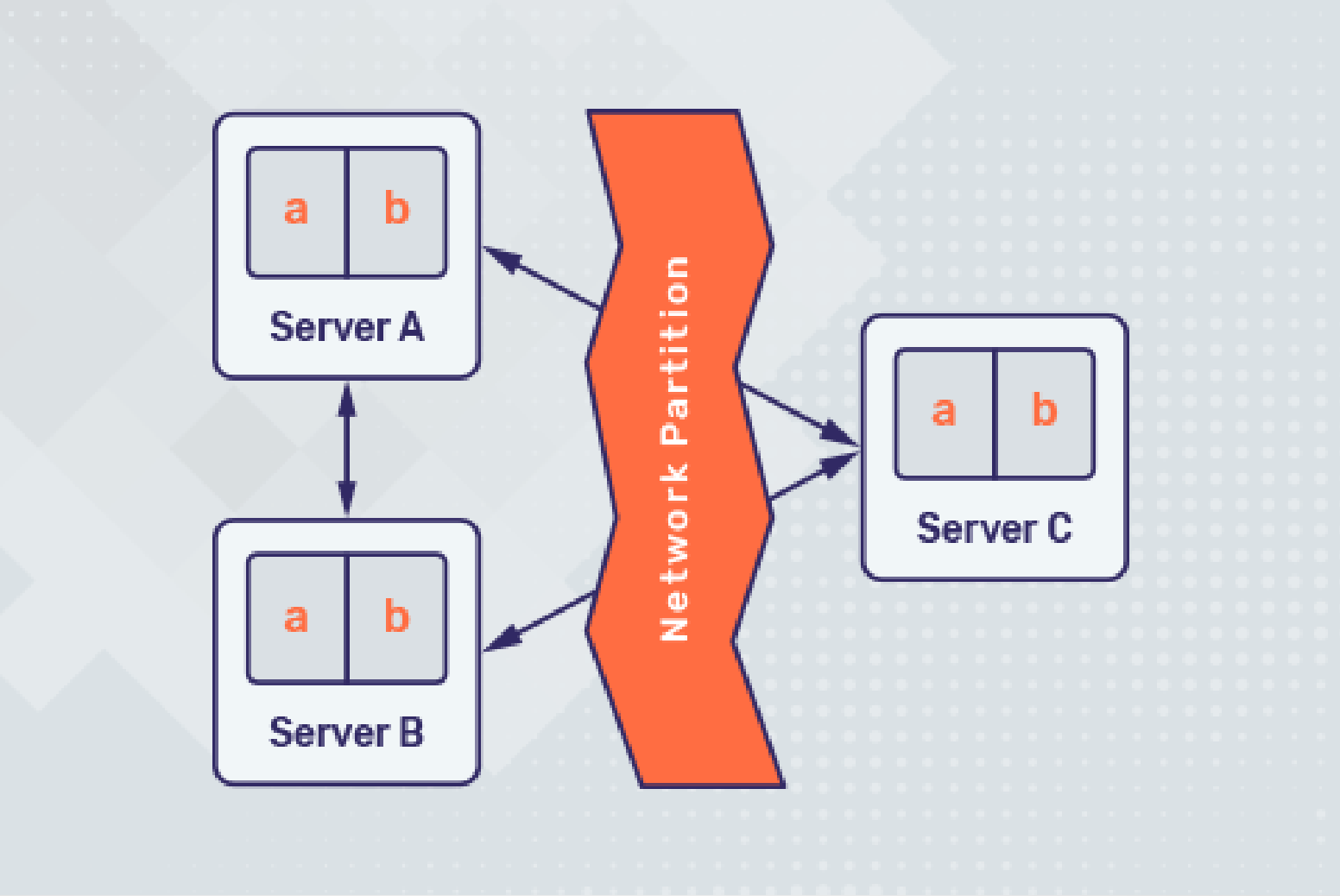
HS1

CUSTOMER ID	FIRST NAME	LAST NAME	CITY
1	Alice	Anderson	Austin
2	Bob	Best	Boston

HS2

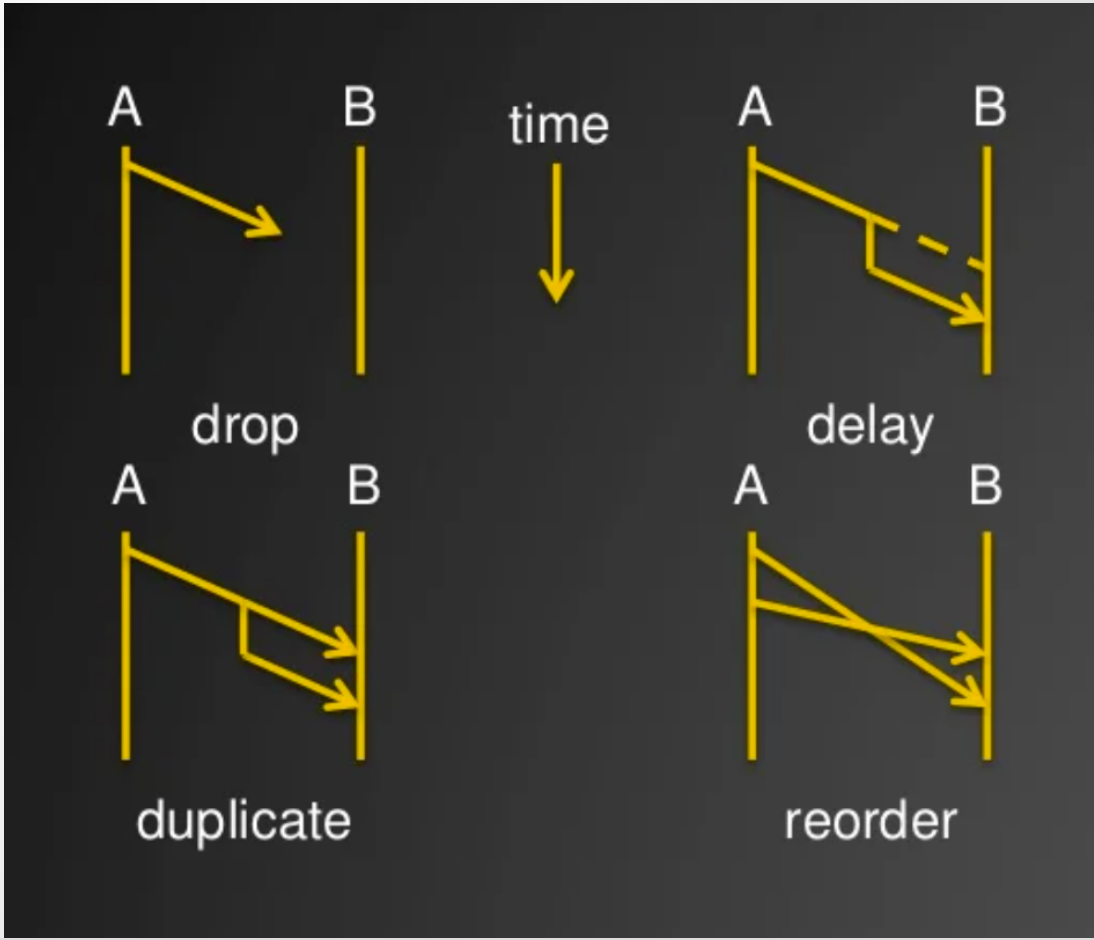
CUSTOMER ID	FIRST NAME	LAST NAME	CITY
3	Carrie	Conway	Chicago
4	David	Doe	Denver

Network Partitioning



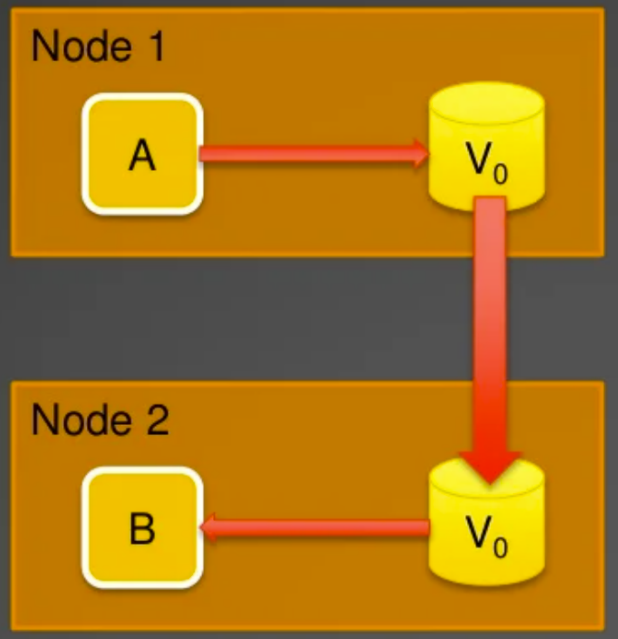
Happens all the time on TCP networks...

For a client drop and delay are the same...

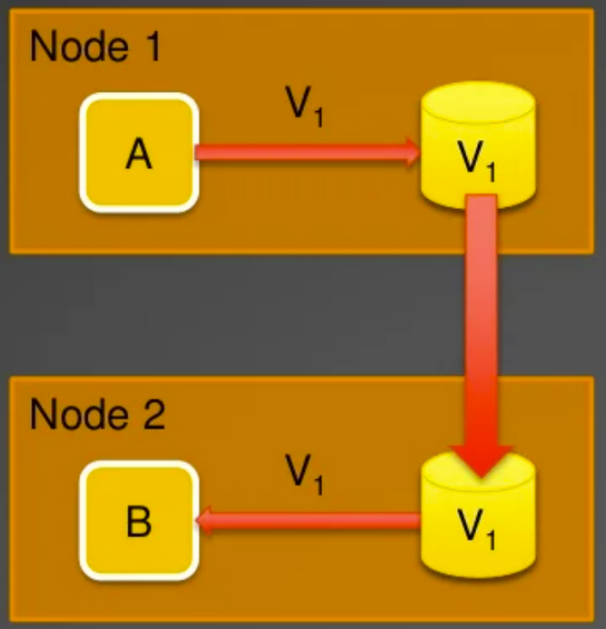


Tradeoff example...

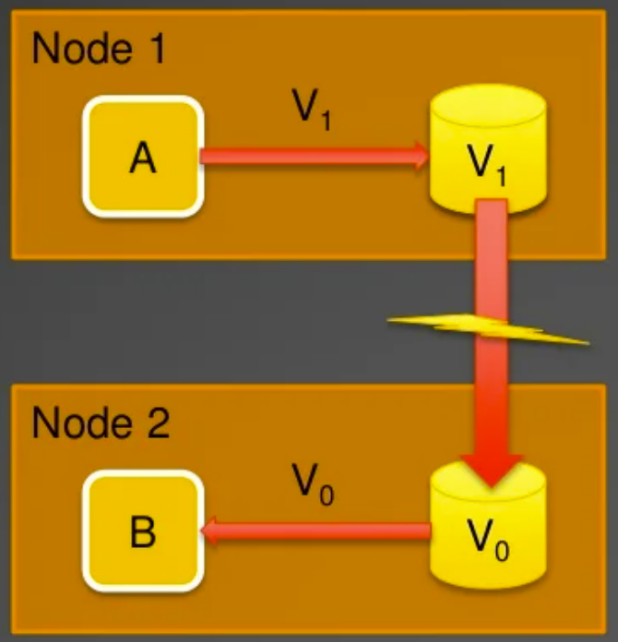
- Let's consider a simple system:
  - Service A *writes* values
  - Service B *reads* values
  - Values are replicated between nodes
- These are "ideal" systems
  - Bug-free, predictable



- "Sunny day scenario":
  - A *writes* a new value V_1
  - The value is *replicated* to node 2
  - B *reads* the new value

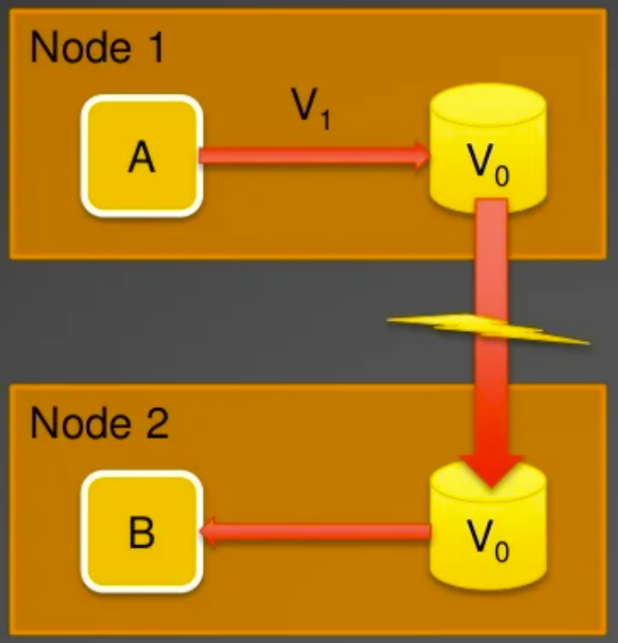


- What happens if the network drops?
  - A *writes* a new value V_1
  - Replication *fails*
  - B still sees the old value
  - The system is *inconsistent*



But you've sent a value nonetheless, so you're available and partition tolerant (**AP**).

- Possible mitigation is *synchronous replication*
  - A writes a new value V_1
  - Cannot replicate, so write is *rejected*
  - Both A and B still see V_0
  - The system is logically *unavailable*



But you've not sent a value, so you're consistent and partition tolerant (**CP**).

Tradeoff example...

If you deliver the 'wrong' value, you're available and partition tolerant (**AP**) but **not consistent**.

If you don't deliver the 'wrong' value, but rather wait for convergence, you're consistent and partition-tolerant (**CP**), but **unavailable**.

CAP

Theorem

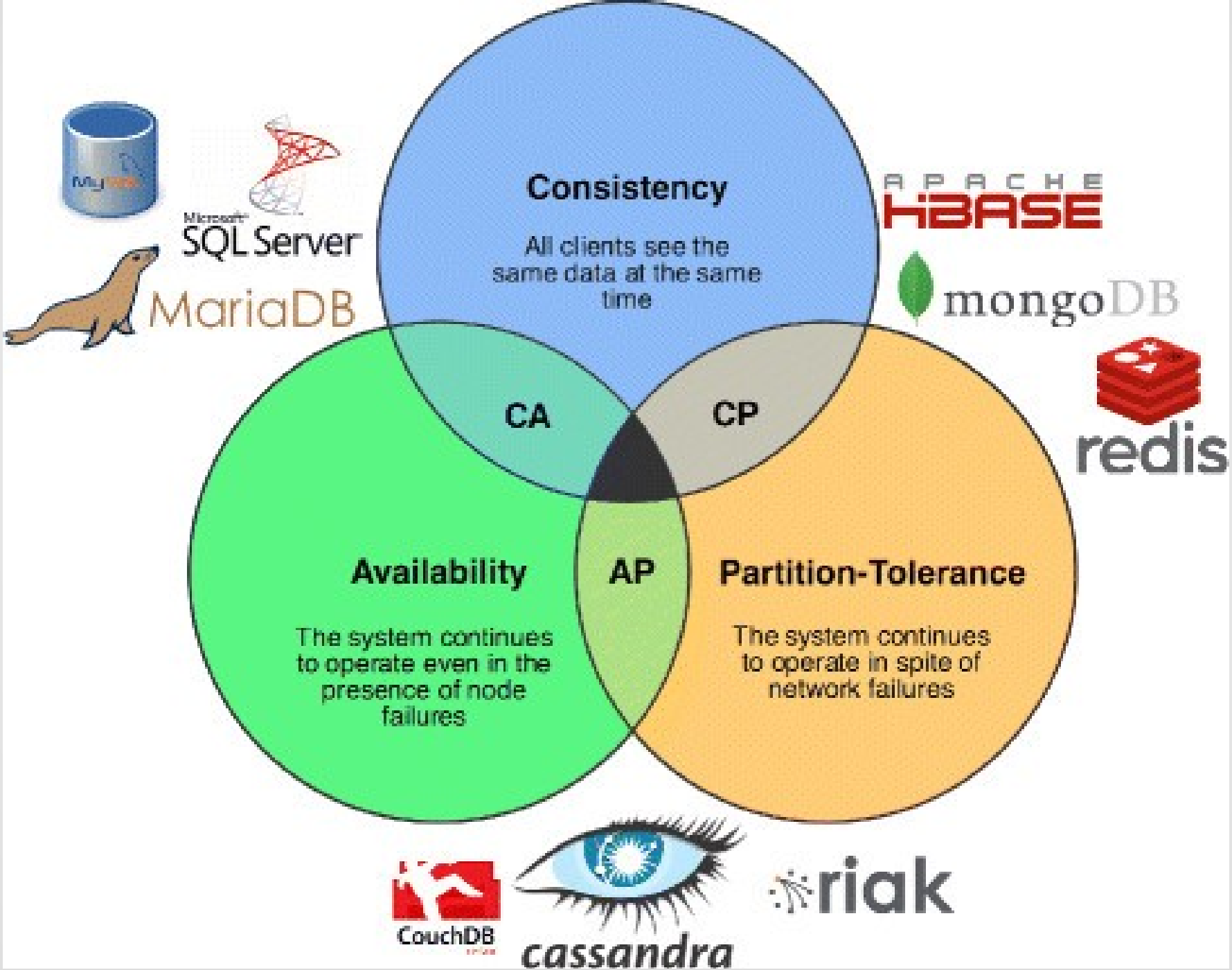
CAP - Theorem

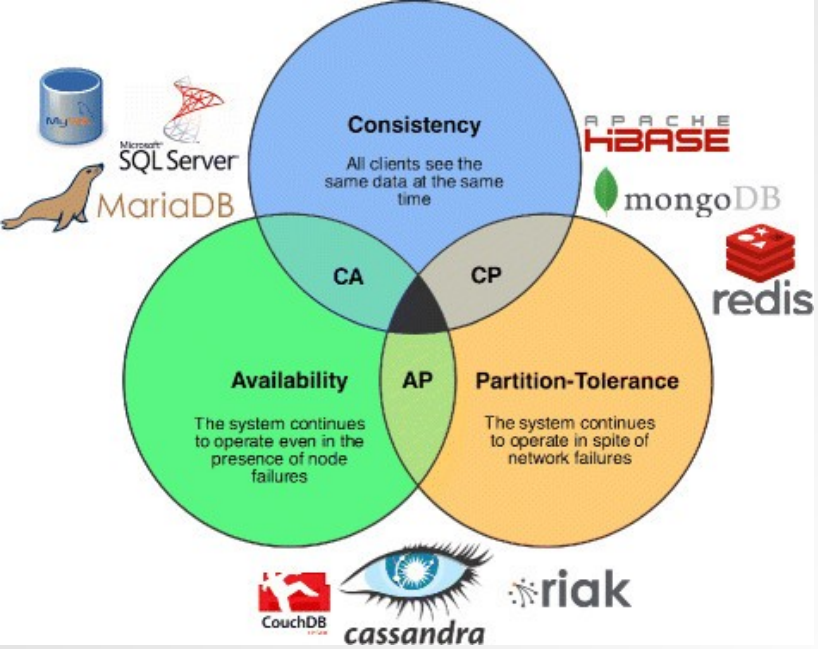
...or Brewer's theorem

- **C onsistency**
- **A vailability**
- **P artition tolerance**

The theorem is...

You may only pick two!


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MySQL
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak



CA (Consistency and Availability):

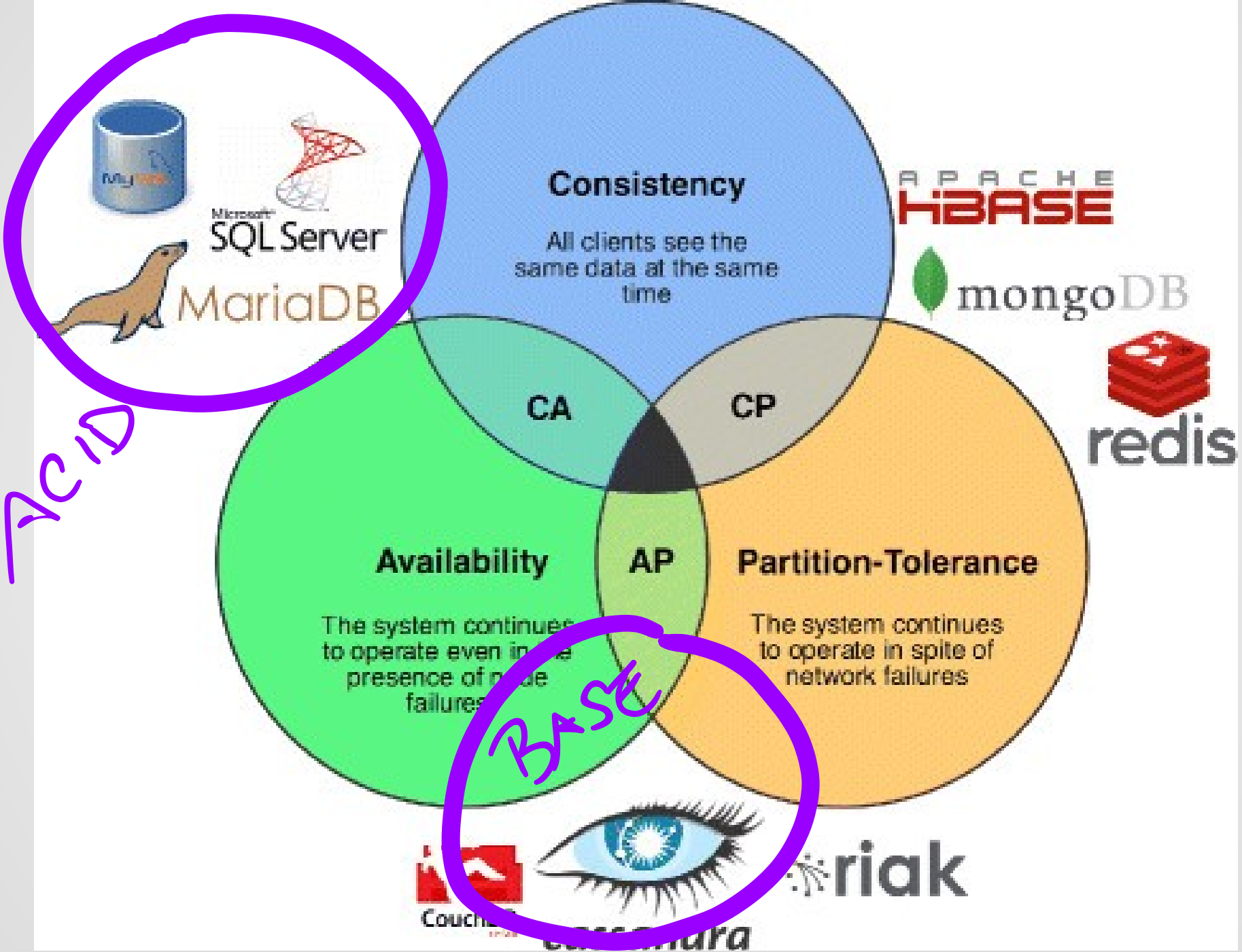
The correct value is delivered at all times, but no resilience to network-partitioning at all. Kafka is CA.

AP (Availability and Partition tolerance):

...tries to return the most recent available version of the information even if it cannot guarantee it is up to date due to network partitioning.

CP (Consistency and Partition tolerance):

...returns an error or time-out if particular information cannot be updated to other nodes due to network partition or failures.


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
ACID
BASE
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
riak

ACID

Atomicity

Consistency

Isolation

Durability

ACID

In 1983, Andreas Reuter and Theo Härder coined the acronym *ACID*, building on earlier work by Jim Gray.

It describes the four properties that are the major guarantees of the transaction paradigm.

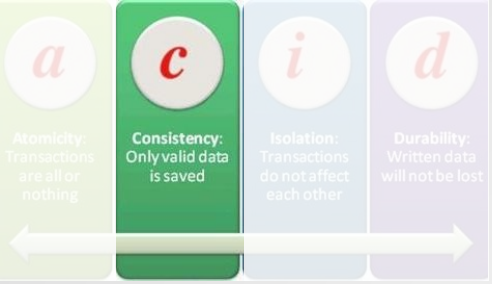
According to Gray and Reuter, the IBM Information Management System supported ACID transactions as early as 1973 (although the acronym was created later).


a
c
i
d
Atomicity: Transactions are all or nothing
Consistency: Only valid data is saved
Isolation: Transactions do not affect each other
Durability: Written data will not be lost

Atomicity

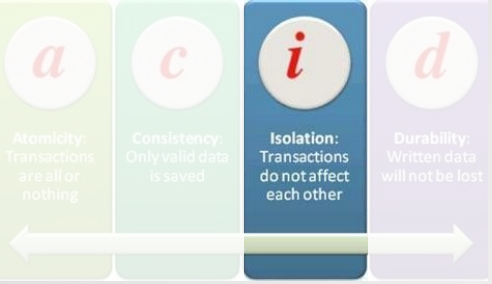
An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes.

Transactions consist of statements and atomicity guarantees that a **transaction is either fully committed or not committed at all**.



Consistency

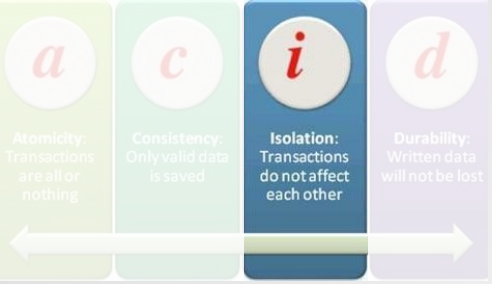
Consistency ensures that a **transaction can only bring the database from one valid state to another**.



Isolation

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time).

Isolation ensures that **concurrent** execution of transactions leaves the database in the **same state** that would have been obtained if the transactions were executed **sequentially**.



Isolation

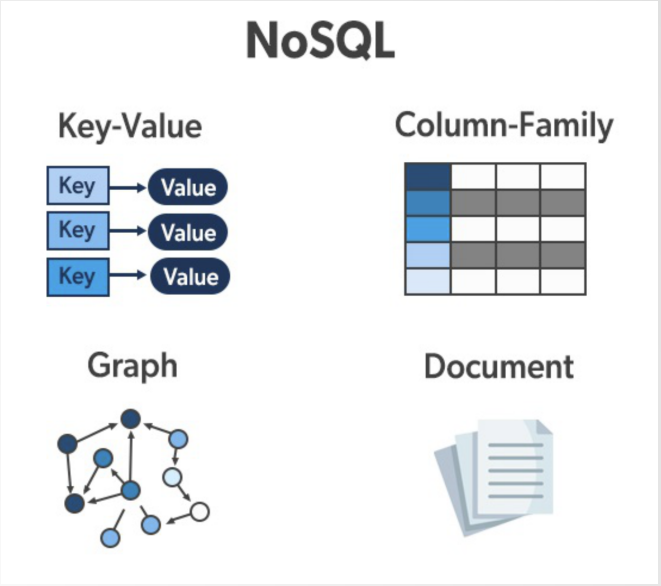
Isolation is the main goal of **concurrency control**; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions.

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash).

This usually means that completed transactions (or their effects) are recorded in **non-volatile memory**.

BASE



Basic **A**vailability

Soft-state

Eventually consistent

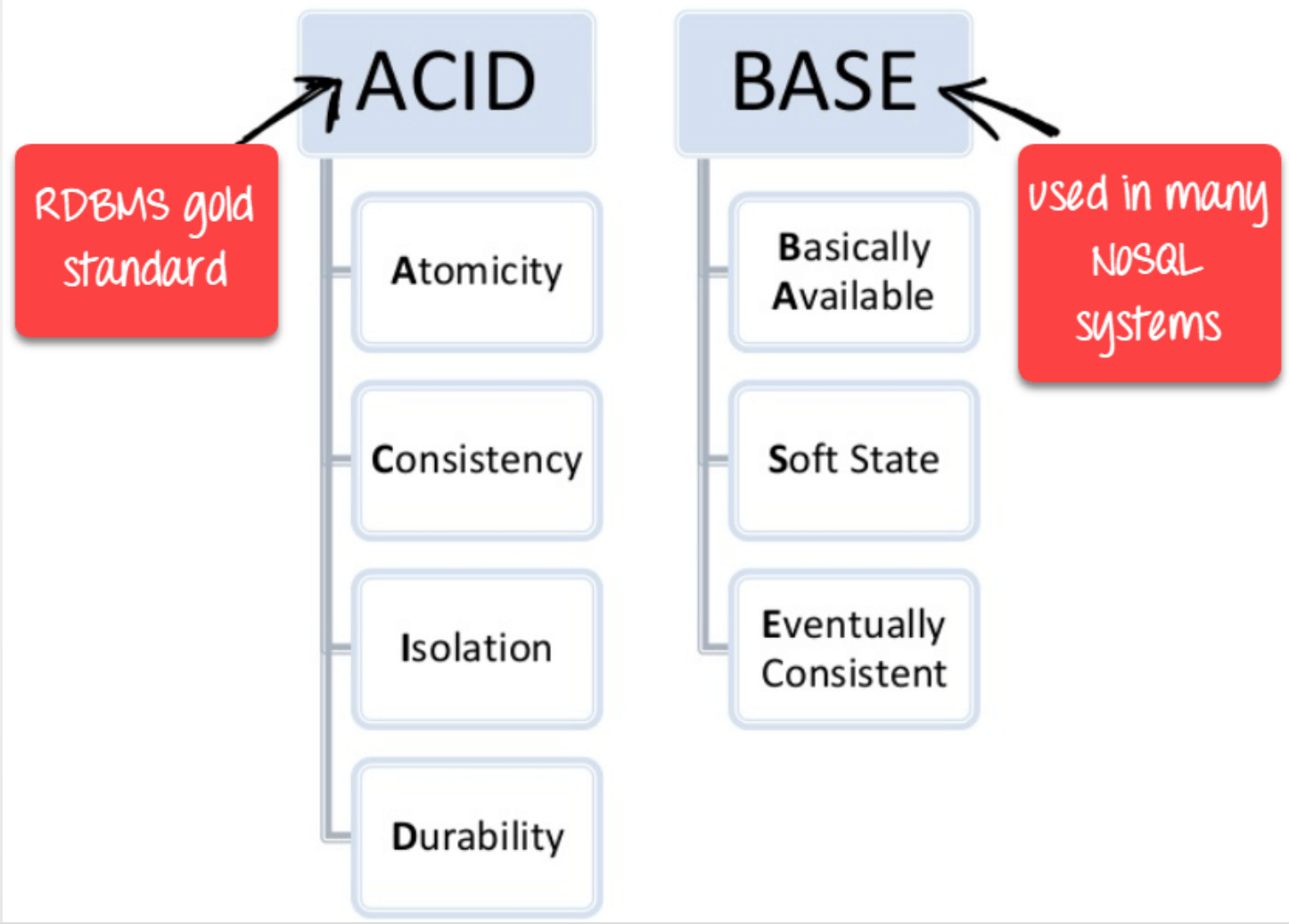
DynamoDB,
Cassandra,
CouchDB,
SimpleDB

Eventual Consistency

Eventual consistency is a consistency model used in distributed computing to achieve high availability.

It informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

BASE


ACID
RDBMS gold standard
Atomicity
Consistency
Isolation
Durability
BASE
used in many NoSQL systems
Basically Available
Soft State
Eventually Consistent

Basically Available

Reading and writing operations are available **as much as possible** (using all nodes of a database cluster), but **might not be consistent**.

(The write might not persist after conflicts are reconciled, and the read might not get the latest write.)

Soft-State

Without consistency guarantees, after some amount of time, we only have some probability of knowing the state, since **it might not yet have converged**.

Eventually Consistent

If we execute some writes and then the system functions long enough, we can know the state of the data.

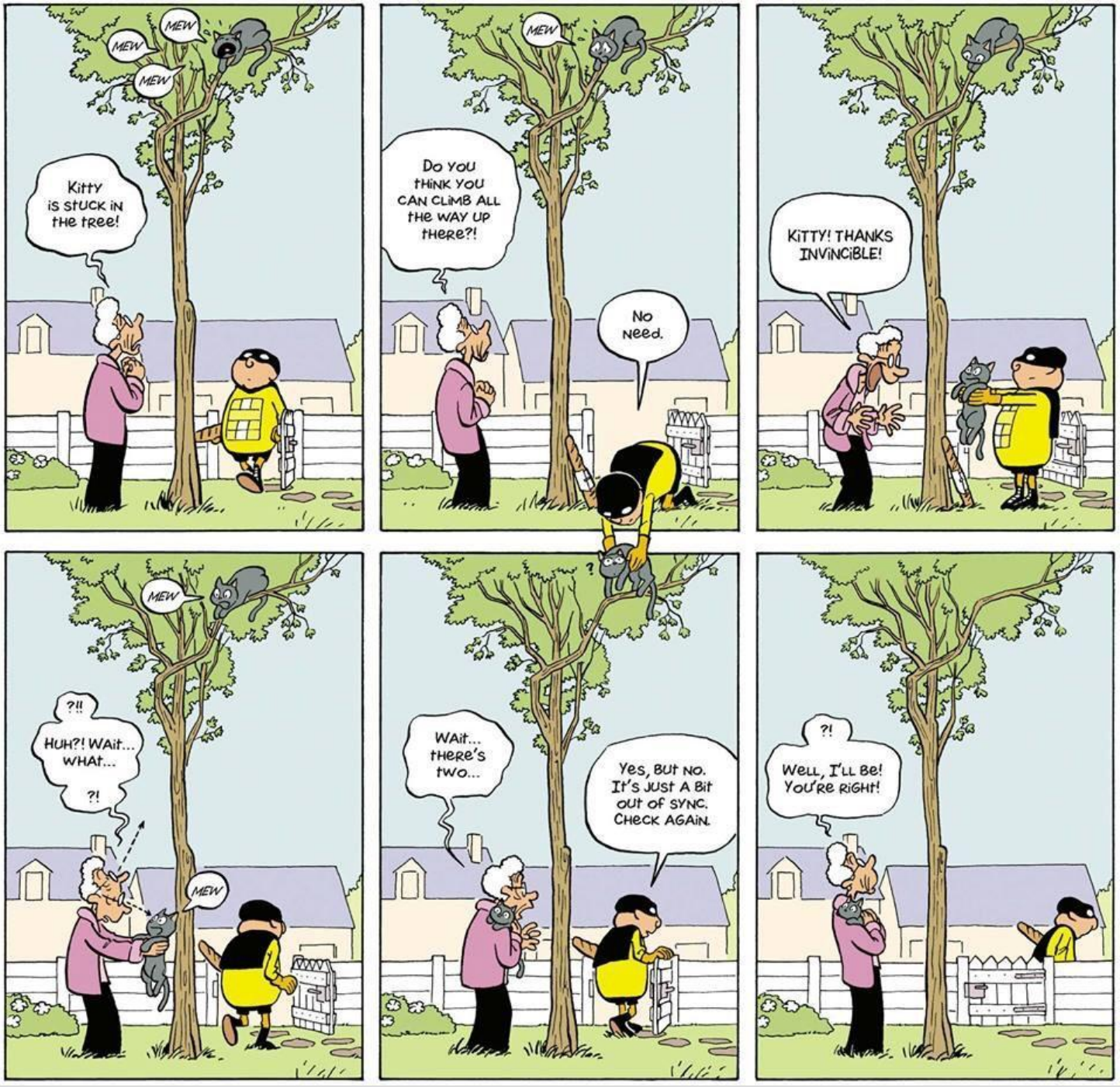
Any further reads of that data item will return the same value.

It is purely a liveness guarantee (reads eventually return the same value) and does not guarantee safety.

An eventually consistent system can return **any value before it converges**.

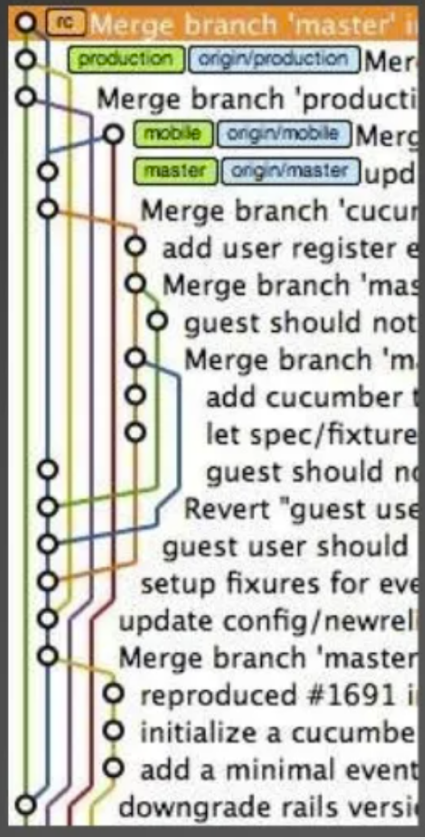
Eventual Consistency

Example


MEW
MEW
MEW
Kitty is stuck in the tree!
MEW
Do you think you can climb all the way up there?!
No need.
KITTY! THANKS INVINCIBLE!
MEW
?!!
HUH?! WAit... WHAt...
?!
MEW
?
WAit... there's two...
Yes, but no. It's just a bit out of sync. Check again.
?!
Well, I'll be! You're right!

Eventual Consistency

- A design which prefers *availability*
- … but guarantees that clients will *eventually* see consistent reads
- Consider *git*:
  - Always available locally
  - Converges via push/pull
  - Human conflict resolution



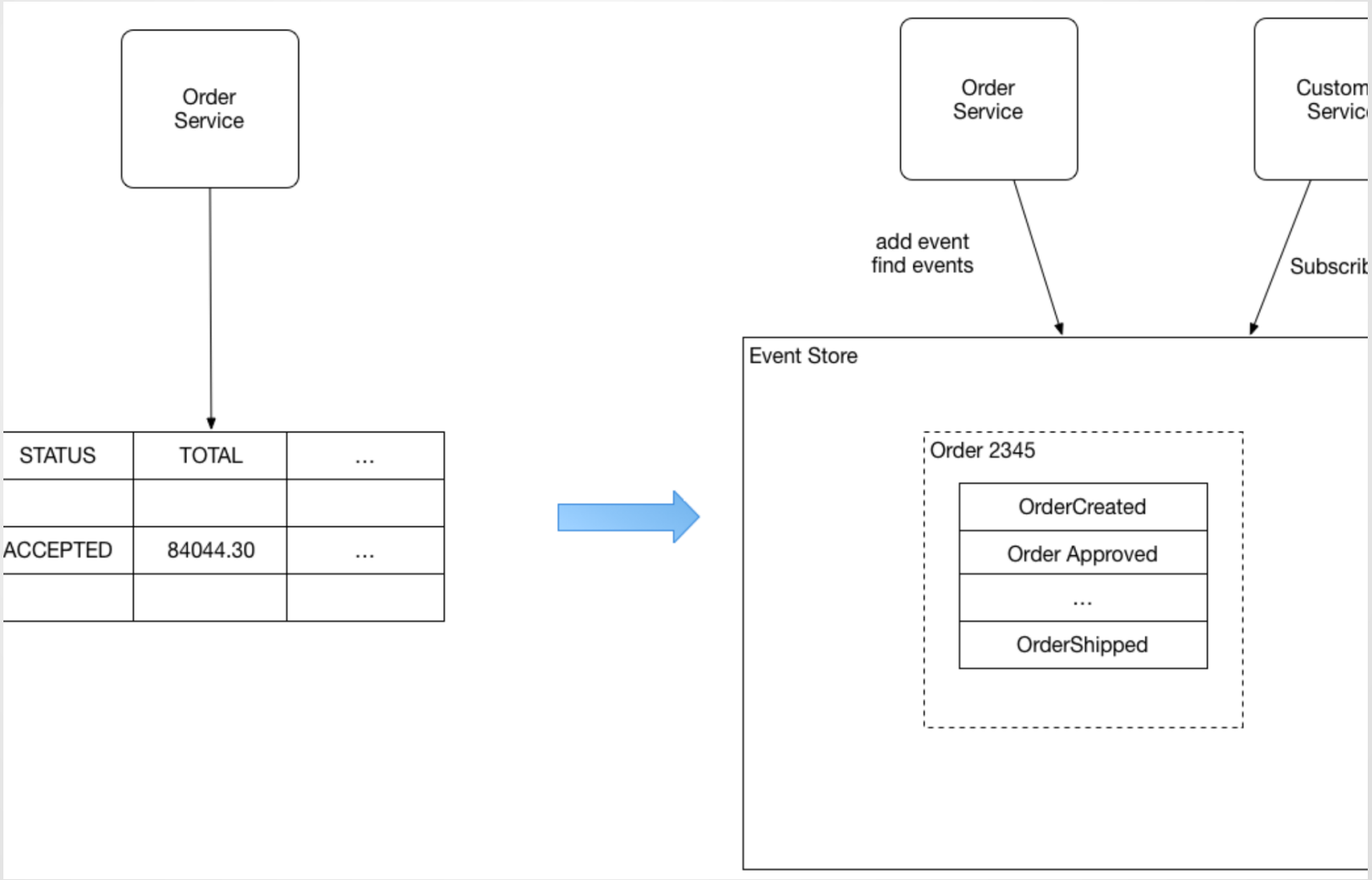
Vector Clocks

- A technique for partial ordering
- Each node has a *logical clock*
  - The clock increases on every *write*
  - Track the last *observed* clocks for each item
  - Include this vector on replication
- When *observed* and *inbound* vectors have no common ancestor, we have a conflict
- This lets us know *when* history diverged

z.B.: Google Docs collaborative editing.

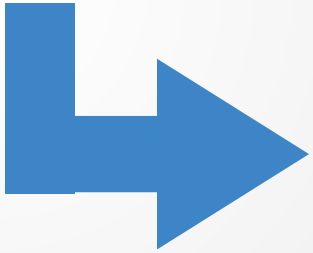
Event Sourcing

Event instead of state



Example

Sku	Quantity	LastReceived	LastShipped
ABC123	59	2020-01-29	
YYZ987	27	2020-12-25	2020-12-31
DER576	18	2020-12-18	2021-01-02



Product Received
Qty=10, Date=2020-01-29

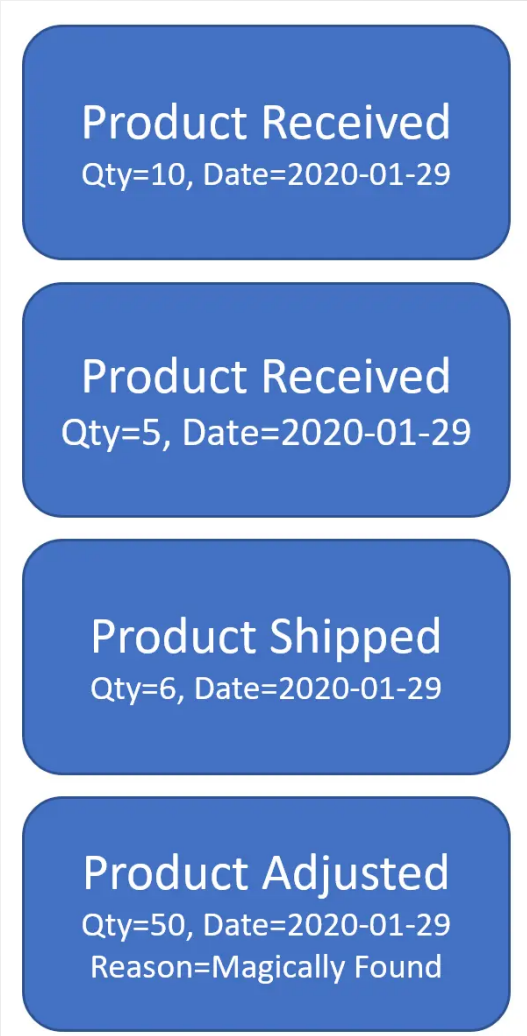
Product Received
Qty=5, Date=2020-01-29

Product Shipped
Qty=6, Date=2020-01-29

Product Adjusted
Qty=50, Date=2020-01-29
Reason=Magically Found

How to do Queries...

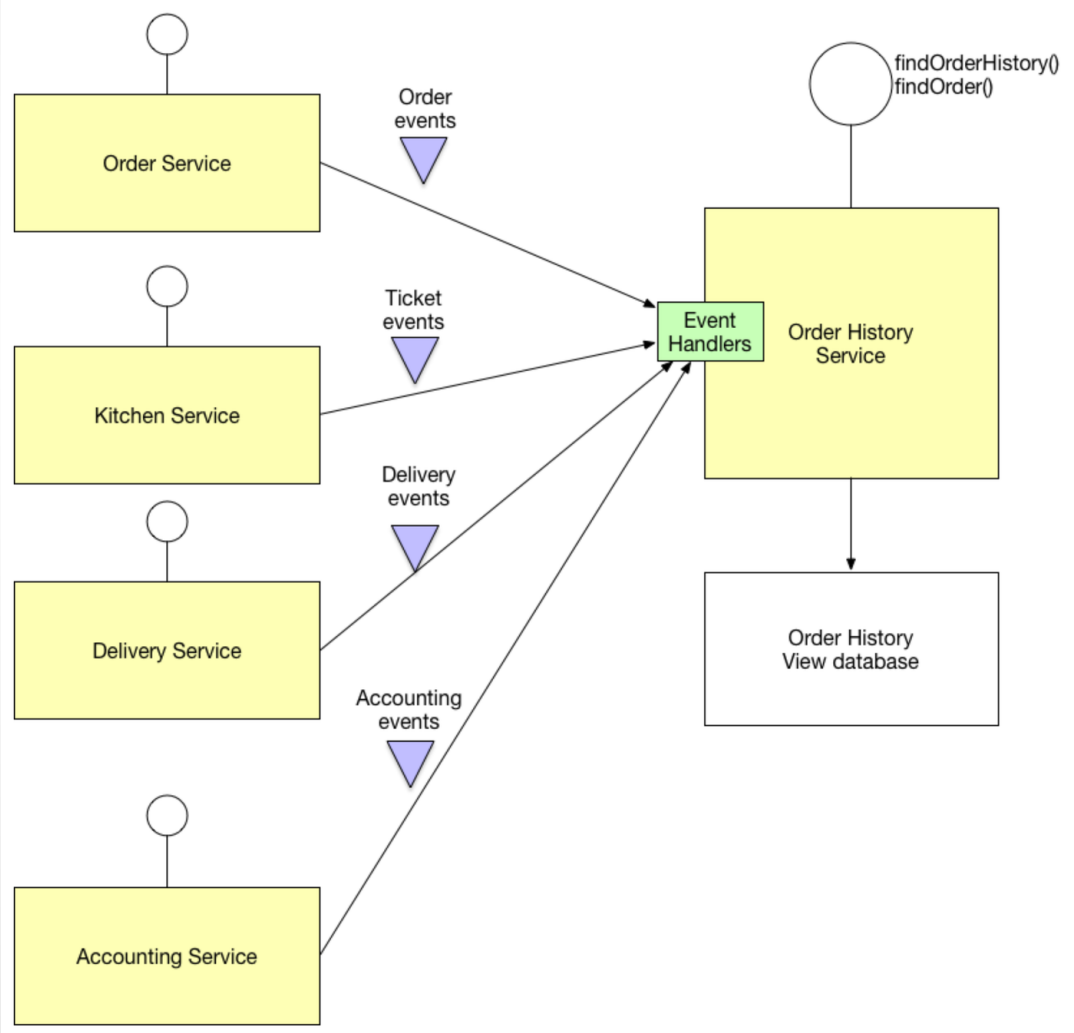
It's complicated to do queries from event data.



How to do Queries...

Solution

Persist event data, converting it into a standard SQL queryable table containing only the current state (like any SQL table would).



References

- https://www.techopedia.com/definition/29165/eventual-consistency
- https://en.wikipedia.org/wiki/ACID
- https://en.wikipedia.org/wiki/Eventual_consistency
- https://systemdesignbasic.wordpress.com/2020/06/20/15-acid-vs-base-database-transactions/
- https://www.linkedin.com/pulse/what-acid-properties-database-aseem-jain/
- https://en.wikipedia.org/wiki/CAP_theorem
- https://www.debadityachakravorty.com/system_design/captheorem/
- https://www.yugabyte.com/blog/achieving-fast-failovers-after-network-partitions-in-a-distributed-sql-database/
- https://codeopinion.com/event-sourcing-example-explained-in-plain-english/
- https://www.slideshare.net/holograph/scaling-out-data-stores-and-the-cap-theorem