THE EVOLUTION OF

CAP THEOREM

TRANSACTION

UNIT OF WORK PERFORMED WITHIN A SYSTEM

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID PROPERTIES

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    - That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

ACID - AN EXAMPLE

Transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

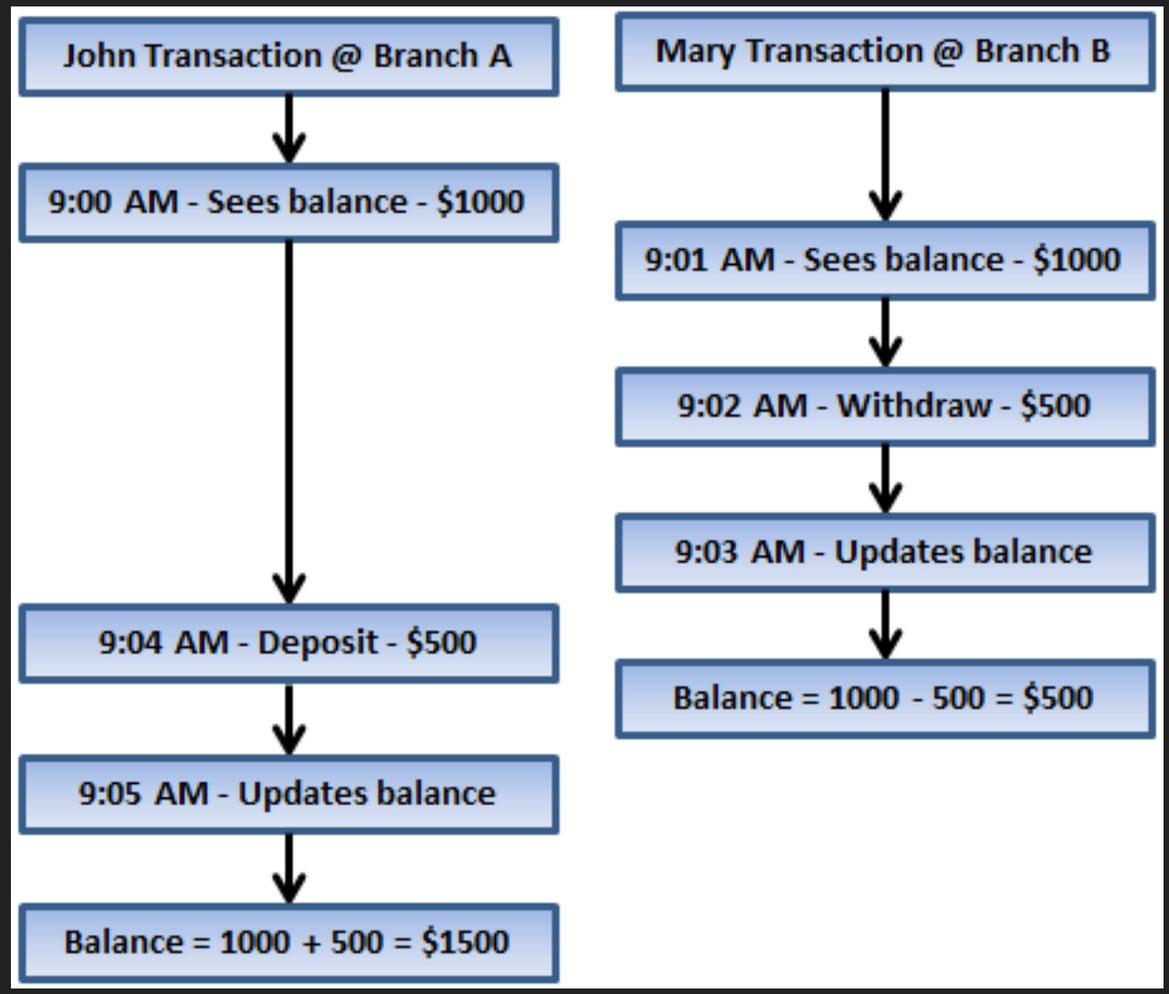
- Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.
- Atomicity requirement – if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result
- Durability requirement – once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement – if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database
  (the sum A + B will be less than it should be).
  Can be ensured trivially by running transactions serially, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

CONCURRENCY

Many users can use (operate on) the data at the same time

Concurrency Advantages:

- Increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
- Reduced average response time for transactions: short transactions need not wait behind long ones.



- **Is there a problem?**

CONCURRENCY

Multiple transactions are allowed to run concurrently in the system.

Concurrency Management:

- Concurrency control schemes – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- Schedules – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

SERIALIZABILITY

Serializability ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations.

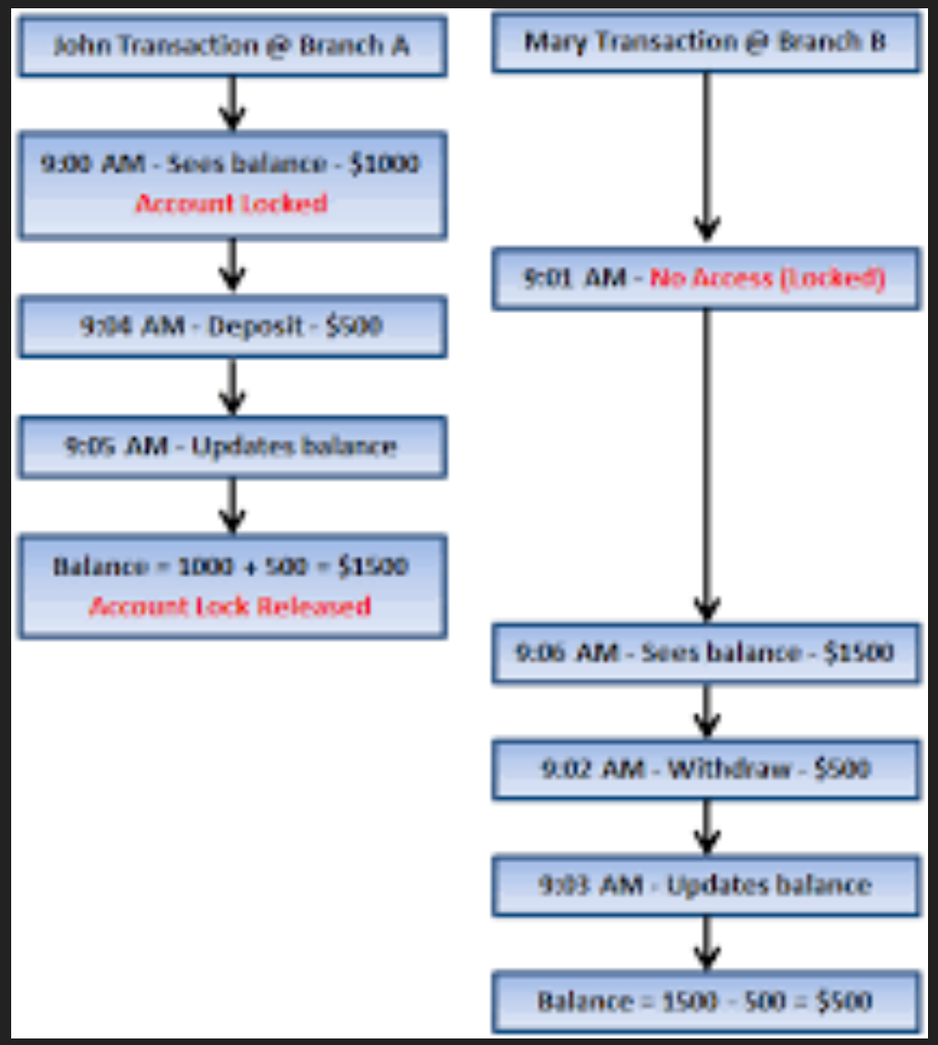
- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

    1. conflict serializability
    2. view serializability

CONCURRENCY

Process of managing simultaneous execution of transactions in a shared **data store**, to ensure the serializability of transactions, is known as **concurrency control**.
Simultaneous execution of transactions over a shared **data store** can create several data integrity and consistency problems: **Lost Updates.**

- **Pessimistic Concurrency**
  - LOCK and allow only one
- **Optimistic Concurrency**
  - Allow multiple but throw an error if record is already modified



CONSISTENCY AND ISOLATION LEVELS

Each user sees a consistent view of the data, including visible changes made by the user's own transactions and transactions of other users

- ISOLATION LEVELS
  - READ COMMITTED
  - READ UNCOMMITTED
  - READ REPEATABLE
  - SERIALIZABLE (Mostly default)

CAP THEOREM - THE INITIAL DEFINITION (2000)

The theorem is a set of basic requirements that describe any distributed system (not just storage/ database systems).

Consistency - All the servers in the system will have the same data so anyone using the system will get the same copy regardless of which server answers their request.

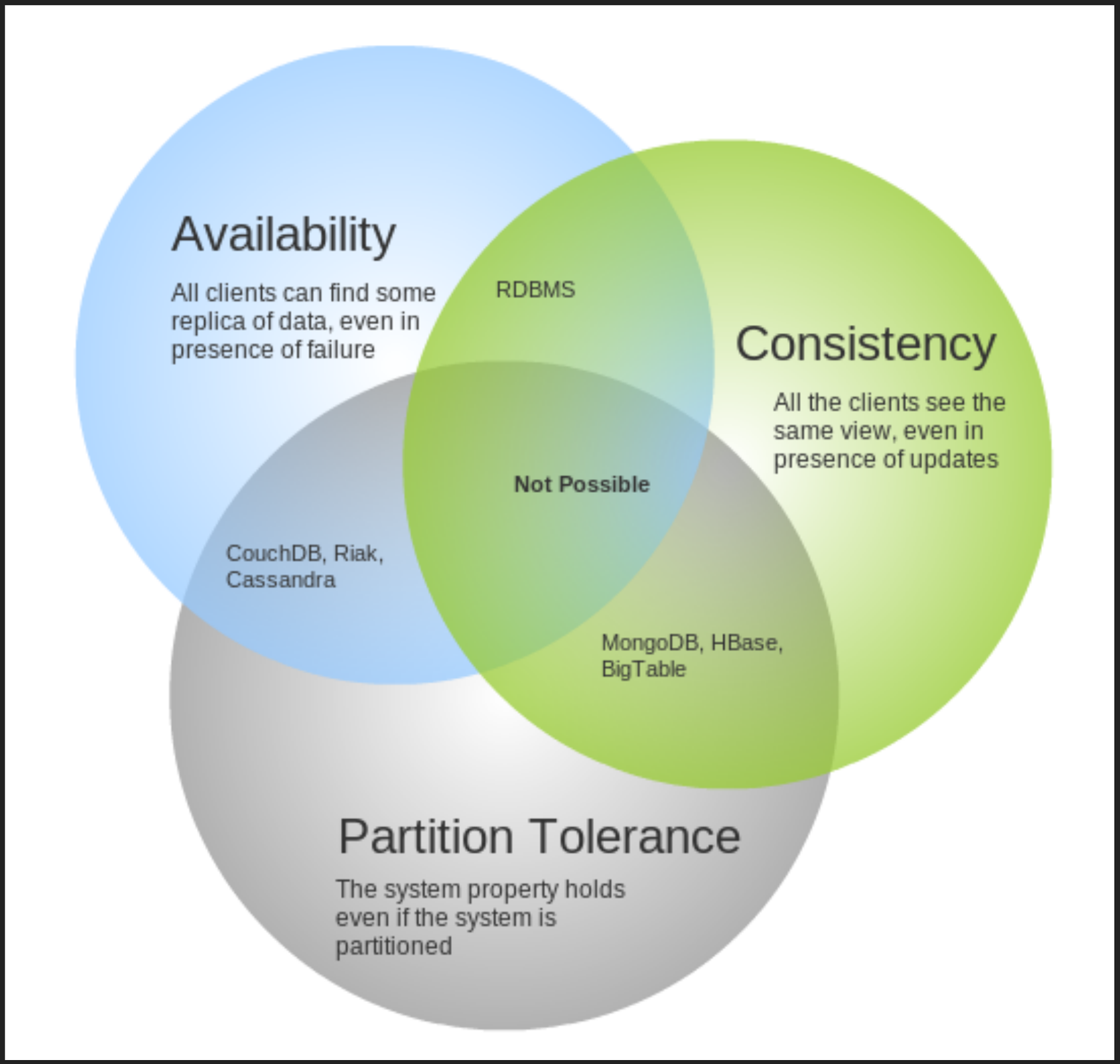
Availability - Every request received by a non-failing [database] node in the system must result in a [non-error] response". It's not sufficient for *some* node to be able to handle the request: *any* non-failing node needs to be able to handle it.

Partition Tolerance - The system continues to operate as a whole even if individual servers fail or can't be reached. (the system partitions into multiple network segments)

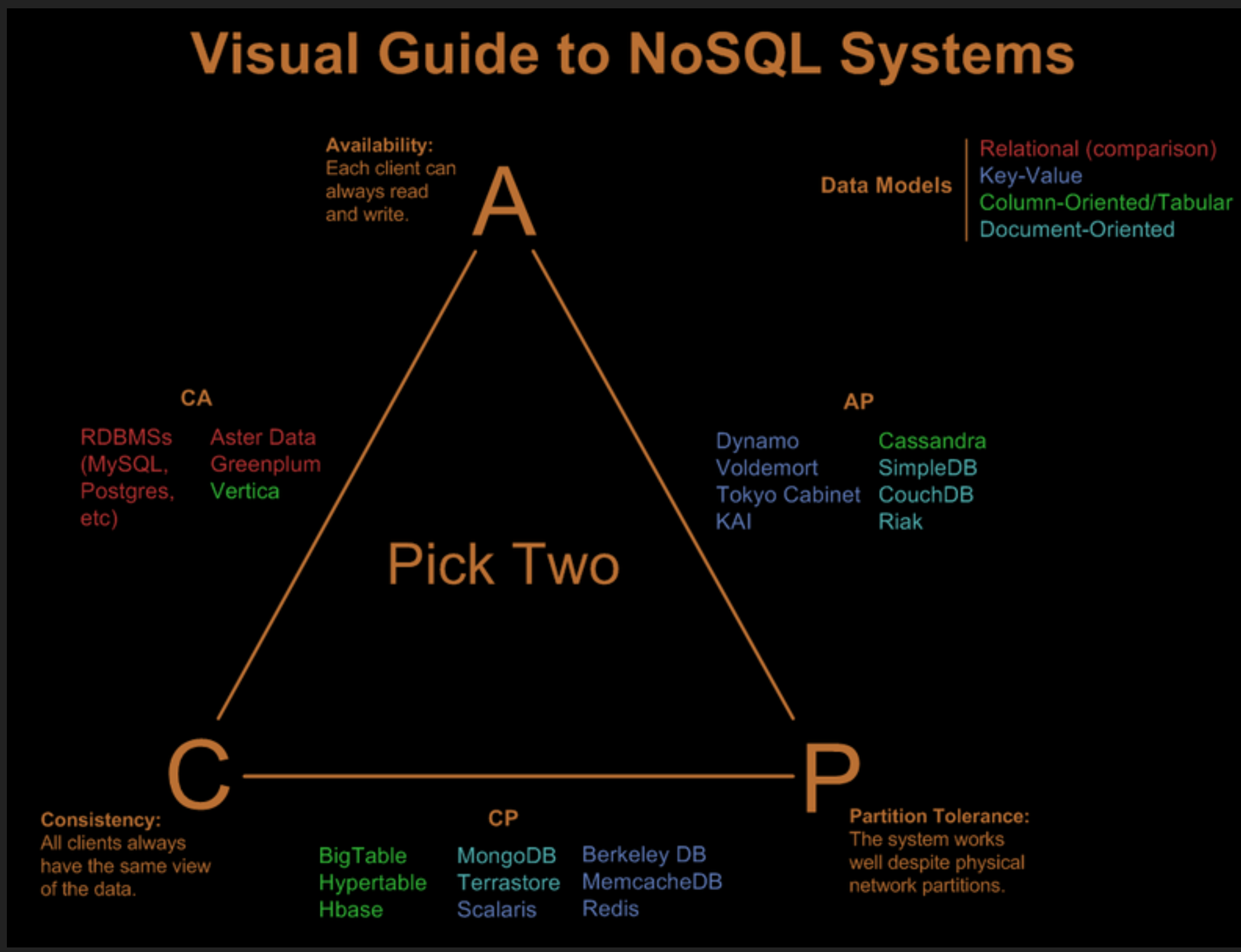
It's theoretically **impossible to have all 3 requirements** met, so a combination of 2 must be chosen and this is usually the deciding factor in what technology is used. a quote here.

Eric Brewer

CAP THEOREM TRADE-OFF



Visual Guide to NoSQL Systems

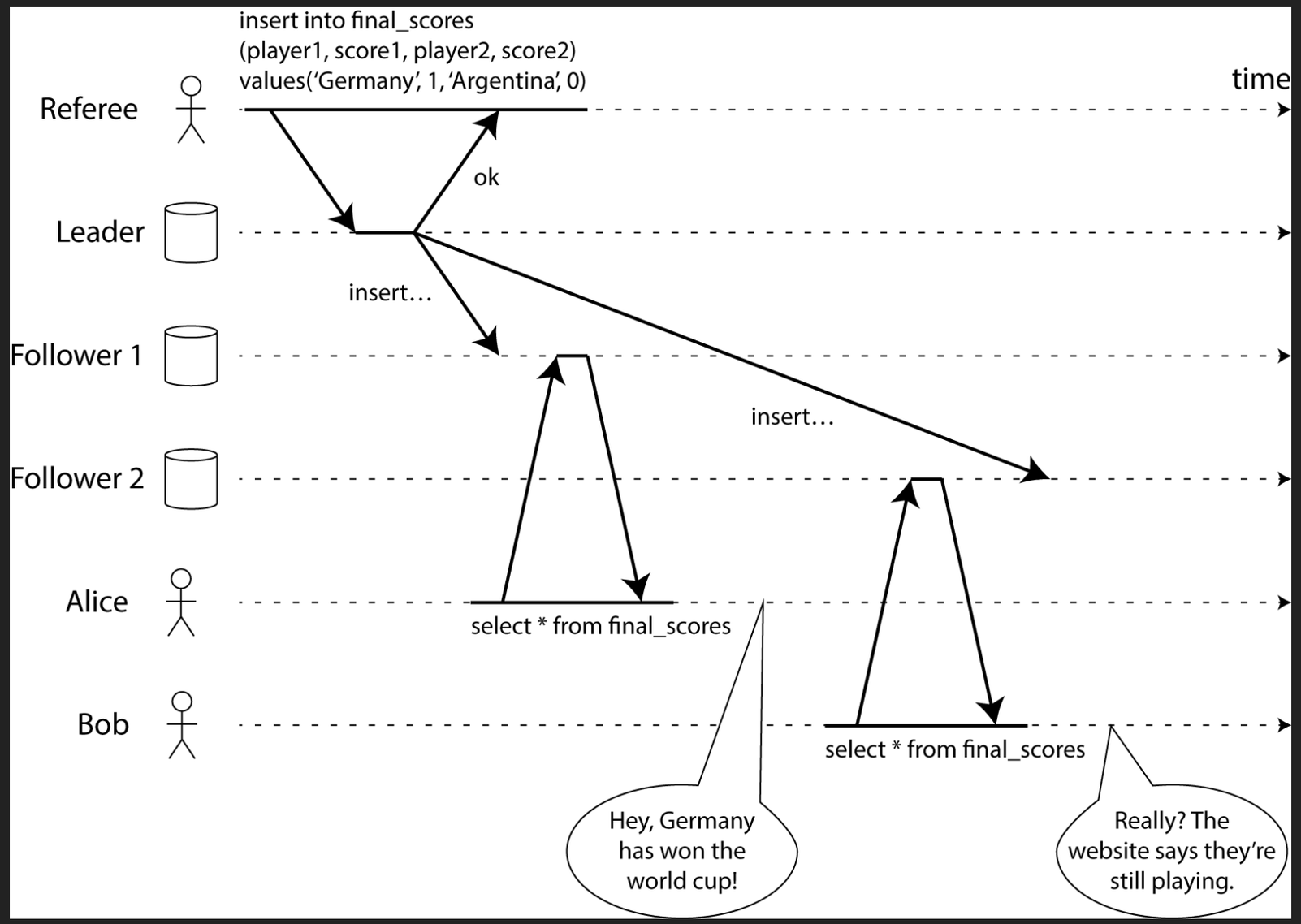


CAP THEOREM - CONSISTENCY

ALL THE SERVERS IN THE SYSTEM WILL HAVE THE SAME DATA - SO ANYONE USING THE SYSTEM WILL GET THE SAME COPY REGARDLESS OF WHICH SERVER ANSWERS THEIR REQUEST.

LINEARIZABILITY:

- If operation B started after operation A successfully completed, then operation B must see the the system in the same state as it was on completion of operation A, or a newer state.



BASE

A BASE SYSTEM GIVES UP ON CONSISTENCY

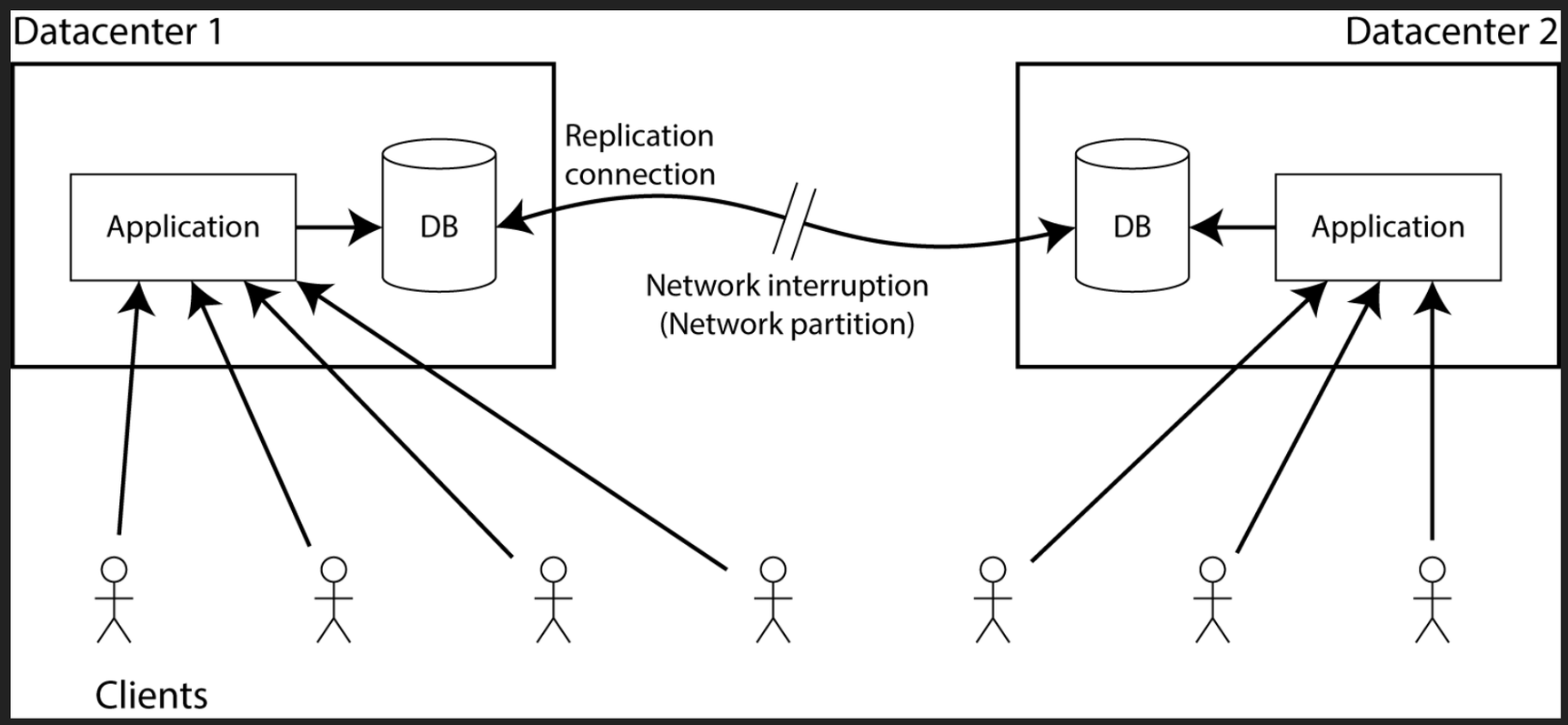
- **Basically available** indicates that the system *does* guarantee availability, in terms of the CAP theorem.
- **Soft state** indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
- **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

The BASE jumper kind is always Basically Available (to new relationships), in a Soft state (none of his relationship last very long) and Eventually consistent (one day he *will* get married).

AVAILABILITY

EVERY REQUEST RECEIVED BY A NON-FAILING NODE MUST RESULT IN A RESPONSE

- Even during severe network failures, the system should terminate and return a response
- A CAP available system should support reads and writes on both sides of the network partition
- This means that the two partitions will not be consistent (non-linearizable)



PARTITION TOLERANCE

THE SYSTEM CONTINUES TO OPERATE AS A WHOLE EVEN IF INDIVIDUAL SERVERS FAIL OR CAN'T BE REACHED

- No set of failures less than total network failure is allowed to cause the system to respond incorrectly
- Detect network partitions, choose between consistency & availability and decide on intrinsic operations that can be allowed/disallowed during partitions

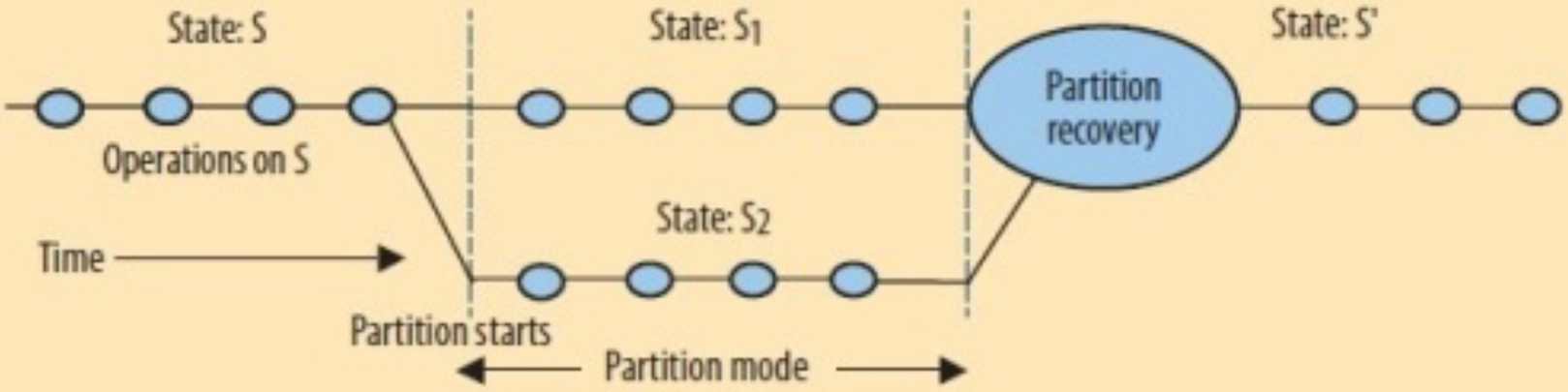


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.