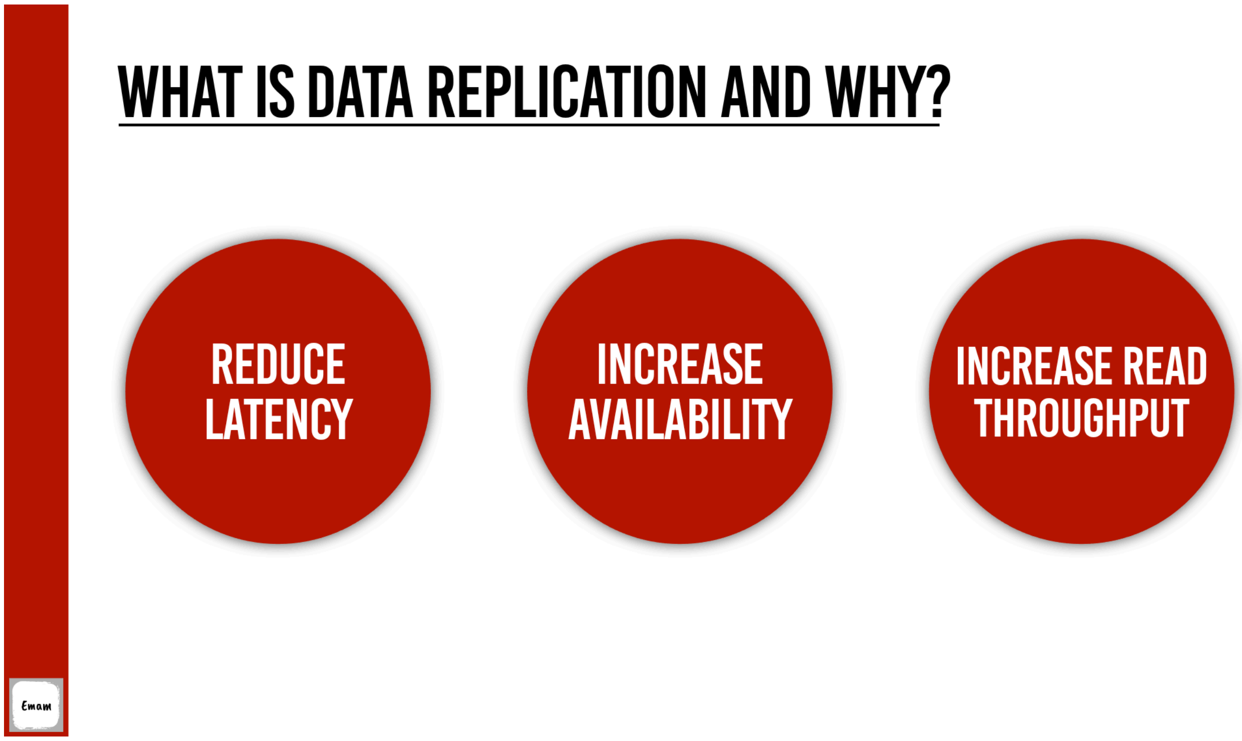
WHAT IS DATA REPLICATION AND WHY?

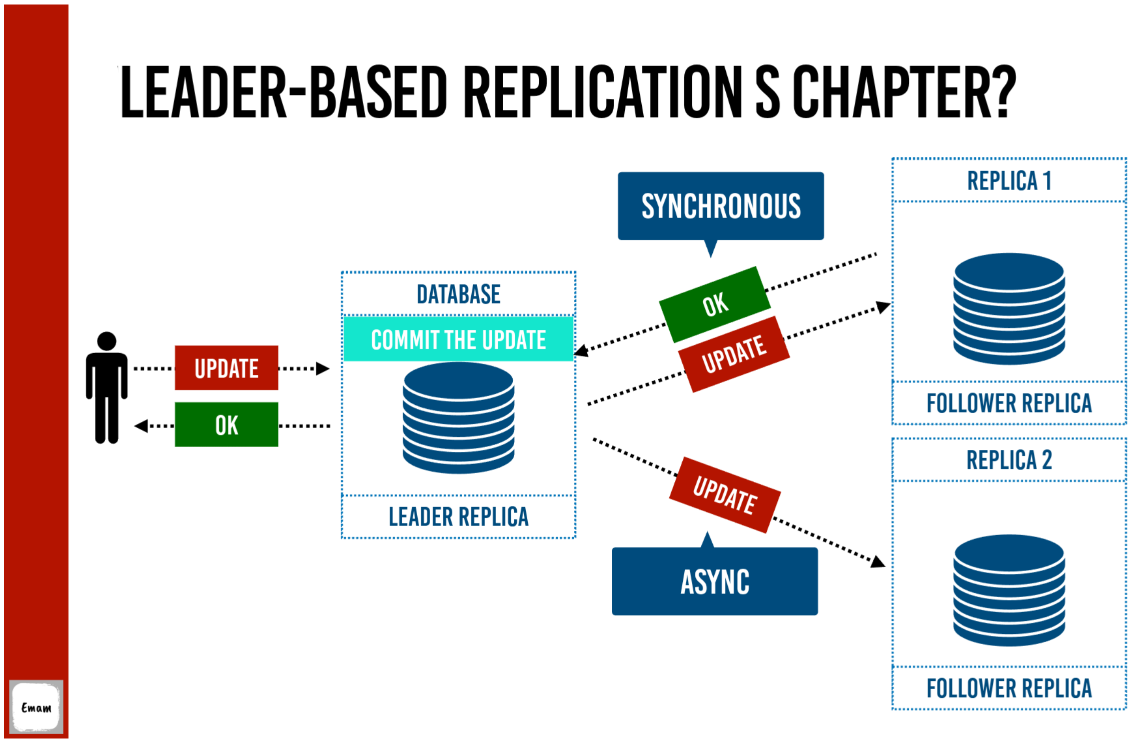
- **Explanation:** Replication = keeping the same data on multiple machines. Three main reasons: • Reduce **latency** (serve users from nearby replica) • Increase **availability** (survive when one machine dies) • Increase **read throughput** (many replicas can answer read queries)
- **Example:** Netflix / YouTube → videos & metadata replicated in many regions → low latency + survive AWS zone failure.
- **Key Takeaway:** • Replication solves latency, availability, and read-scaling — the three biggest reasons to run more than one database node.

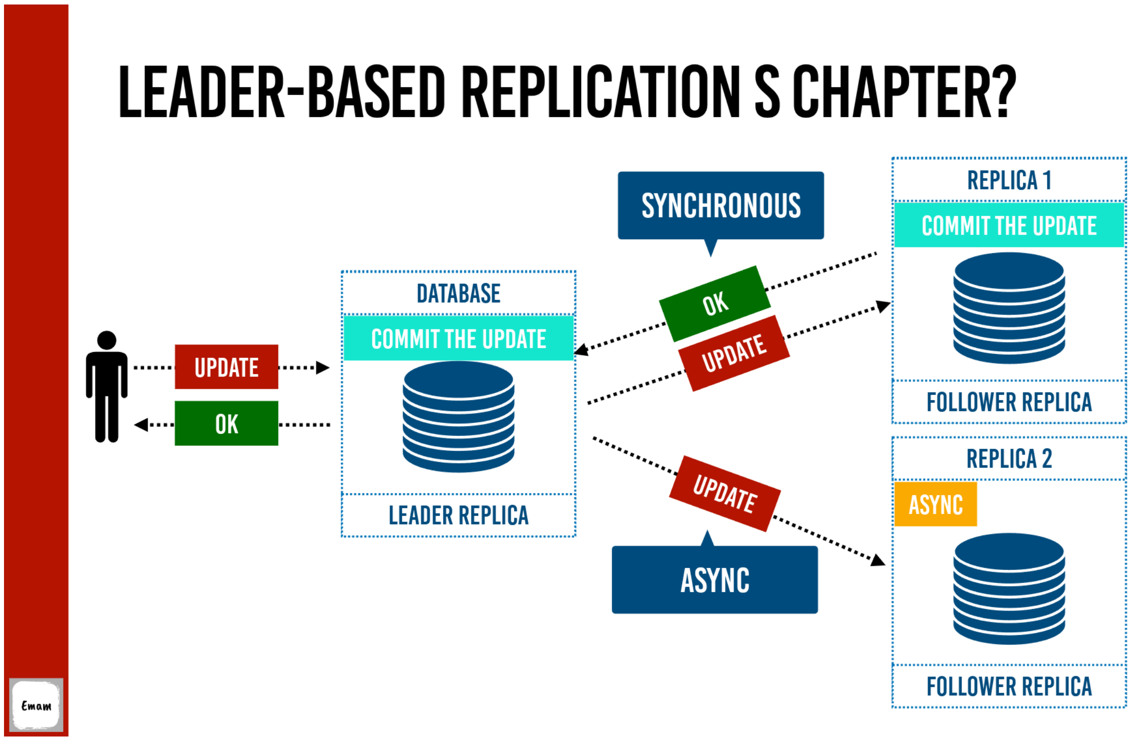


LEADERER-BASED REPLICATION S CHAPTER

- **Explanation:** When the leader accepts a write: • **Synchronous** follower → leader waits until the follower confirms it received & applied the write before telling the client "OK". • **Asynchronous** follower → leader sends the change but does **not** wait — it tells the client "OK" immediately. → Synchronous = stronger durability, but slower & can block writes if follower is slow/down. → Asynchronous = faster, but you can lose recent writes if leader crashes.
- **Example:** PostgreSQL default = asynchronous replication (fast). You can turn on synchronous_commit = on → becomes semi-synchronous (leader waits for at least one synchronous standby → popular for important data).

- **Key Takeaway:** • Almost all production systems use **mostly asynchronous + 1 synchronous standby** (semi-synchronous) — best balance between speed and not losing data when leader dies.



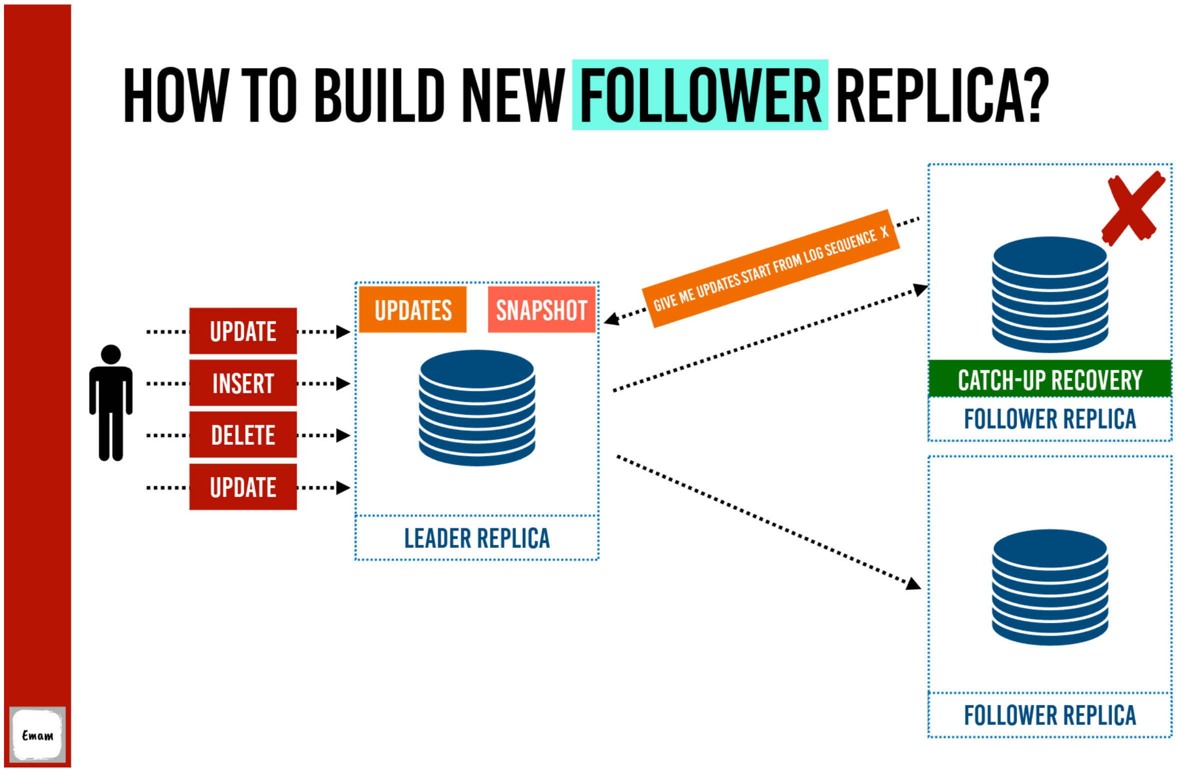


HOW TO BUILD NEW FOLLOWER REPLICA

- **Explanation:** Steps to add a new follower without downtime:
  1. Take consistent snapshot of leader (without locking everything).
  2. Copy snapshot to new follower.

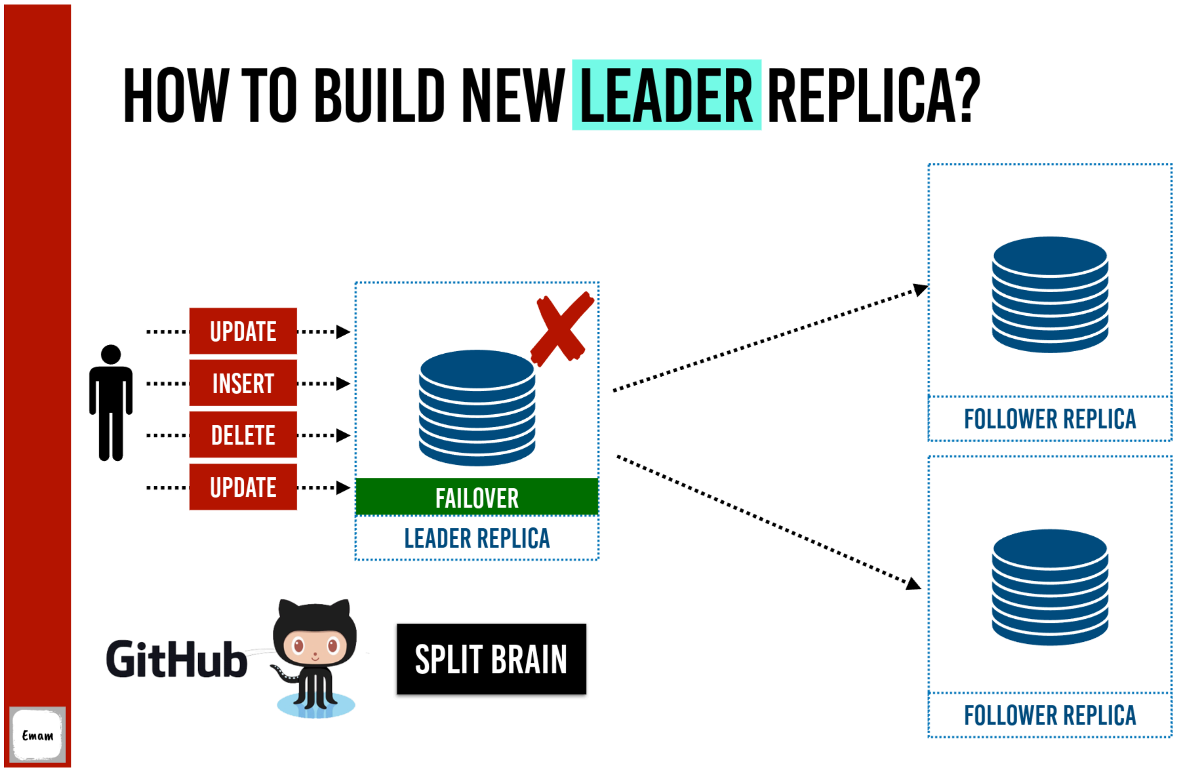
3. Follower asks leader: "give me all changes since this snapshot position".
4. Leader sends the backlog → follower applies it → now caught up and stays in sync.

- **Example:** PostgreSQL pg_basebackup + streaming replication. MySQL → Percona XtraBackup + binlog coordinates.
- **Key Takeaway:** • Modern databases have automated / semi-automated ways to add new replicas without stopping writes — snapshot + catch-up is the standard pattern.



HOW TO BUILD NEW LEADER REPLICA?

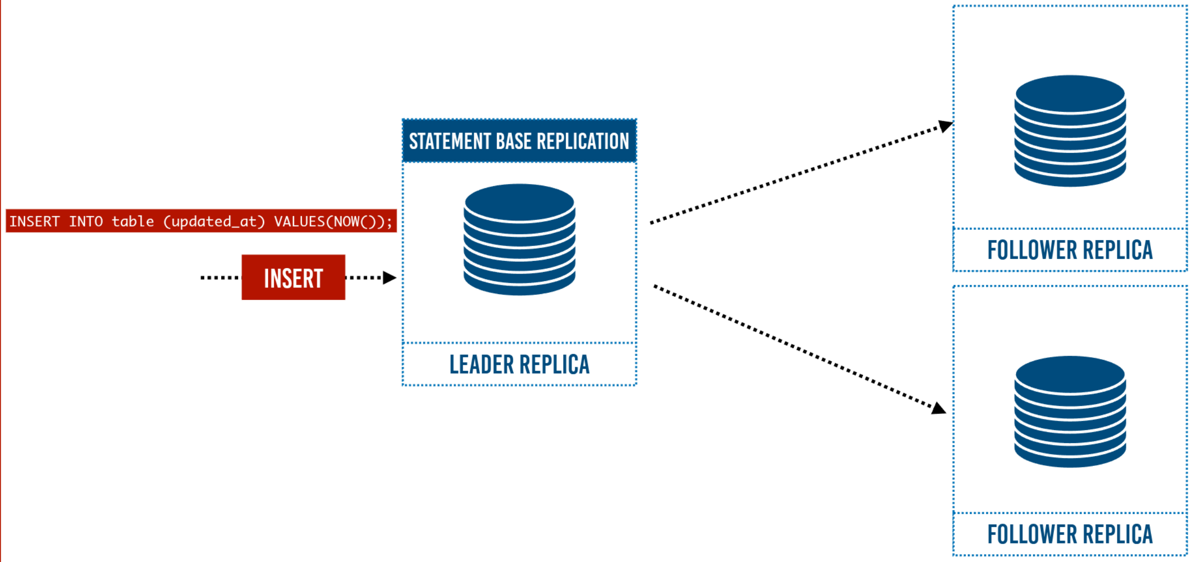
- **Explanation:** When the current leader (primary) database dies or is unreachable, the system must promote one of the followers to become the new leader. But there's a dangerous risk: if the old leader comes back online later and still thinks it's the leader, both nodes might accept writes → this creates **split-brain** (two leaders thinking they are in charge → data corruption).
- **Example:** GitHub in 2012 had a famous split-brain incident during MySQL failover. An old primary came back online, both nodes accepted writes → same auto-increment IDs were reused → private user data leaked between accounts.
- **Key Takeaway:** • Split-brain is one of the most dangerous failure modes in leader-based replication — preventing it is why automatic failover is very hard and many teams prefer manual failover.



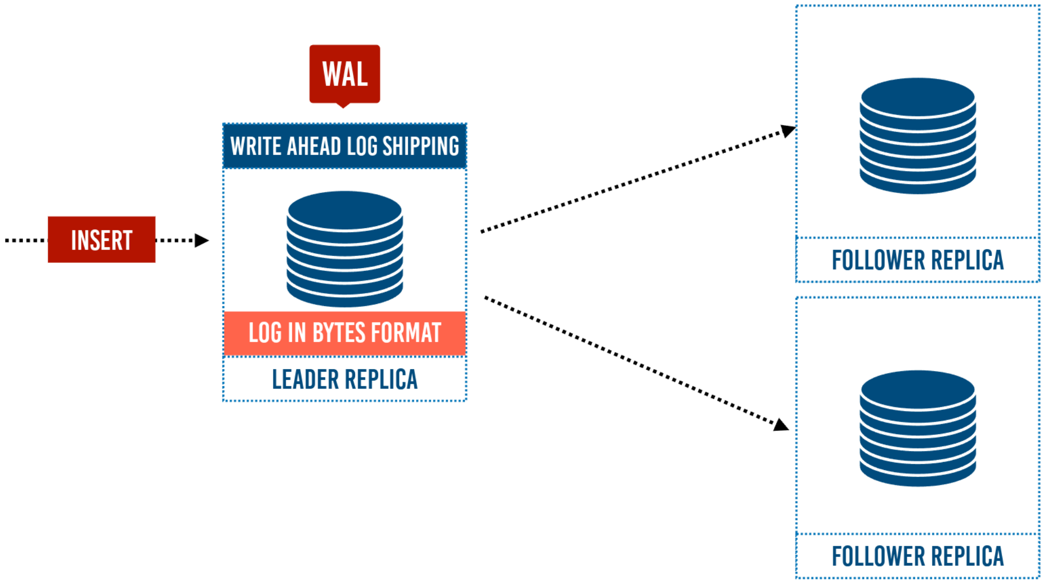
HOW DO LEADER SEND THE UPDATES TO FOLLOWERS

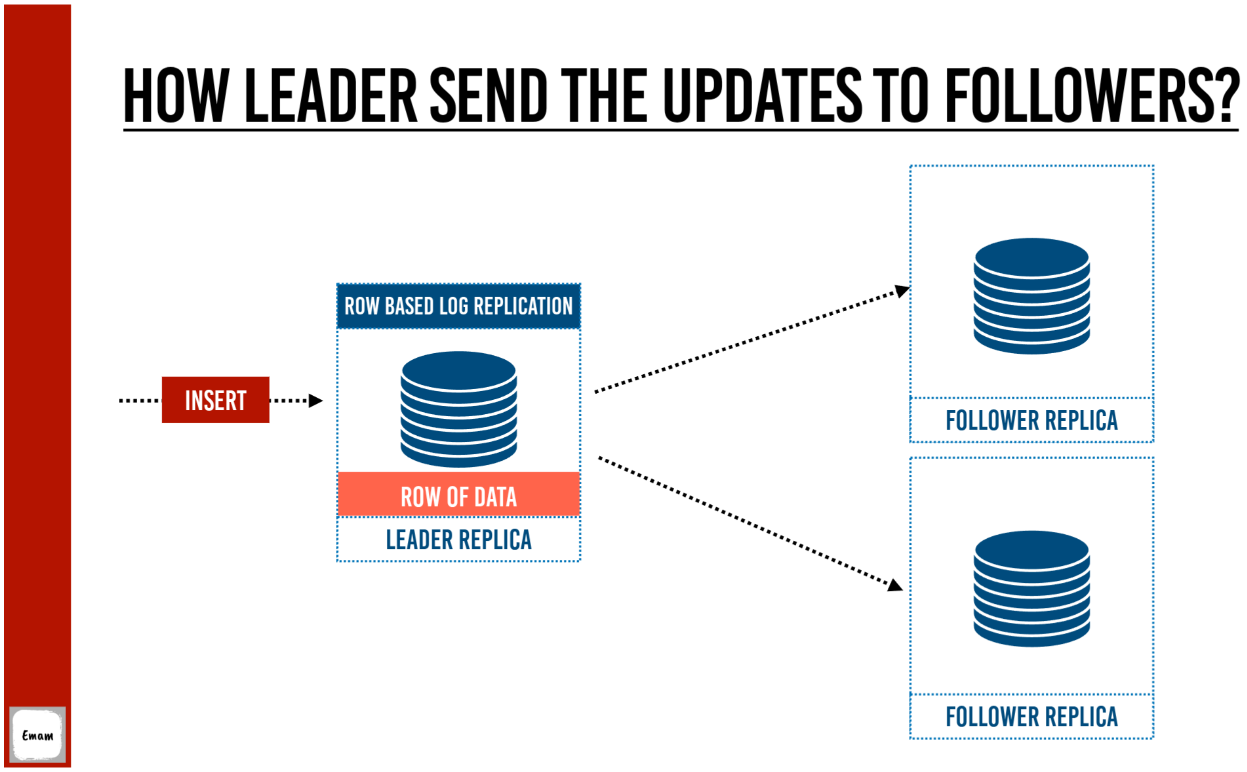
- **Explanation:** The leader doesn't send SQL statements or rows — it sends the **physical Write-Ahead Log (WAL)** or binlog entries (sequence of low-level changes: which bytes changed in which blocks). Followers receive the log and apply exactly the same changes → very efficient and keeps replicas identical.
- **Example:** PostgreSQL → WAL shipping (physical replication). MySQL → binlog (can be statement, row, or mixed format).
- **Key Takeaway:** • WAL / binlog shipping is the most common, battle-tested way leaders send changes — it's fast, reliable, and used by almost every production relational database.

HOW LEADER SEND THE UPDATES TO FOLLOWERS?


INSERT INTO table (updated_at) VALUES(NOW());
INSERT
STATEMENT BASE REPLICATION
LEADER REPLICA
FOLLOWER REPLICA
FOLLOWER REPLICA

HOW LEADER SEND THE UPDATES TO FOLLOWERS?

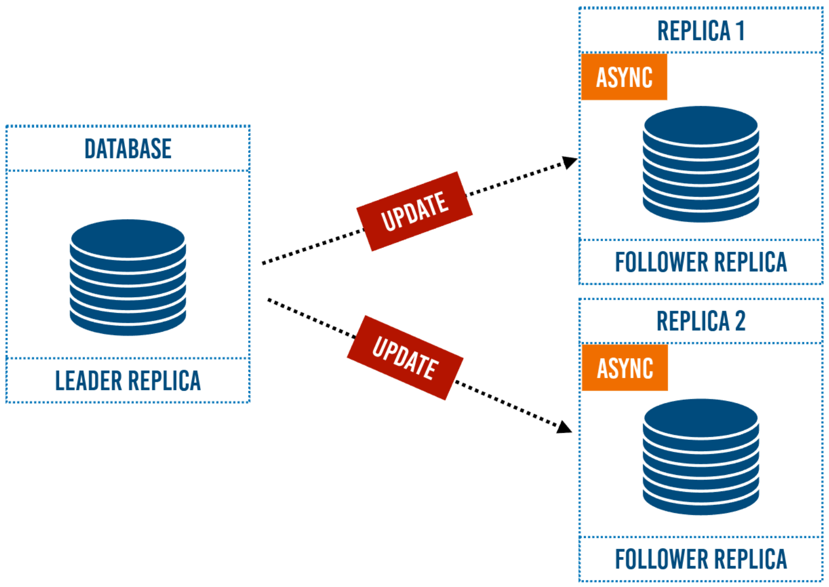




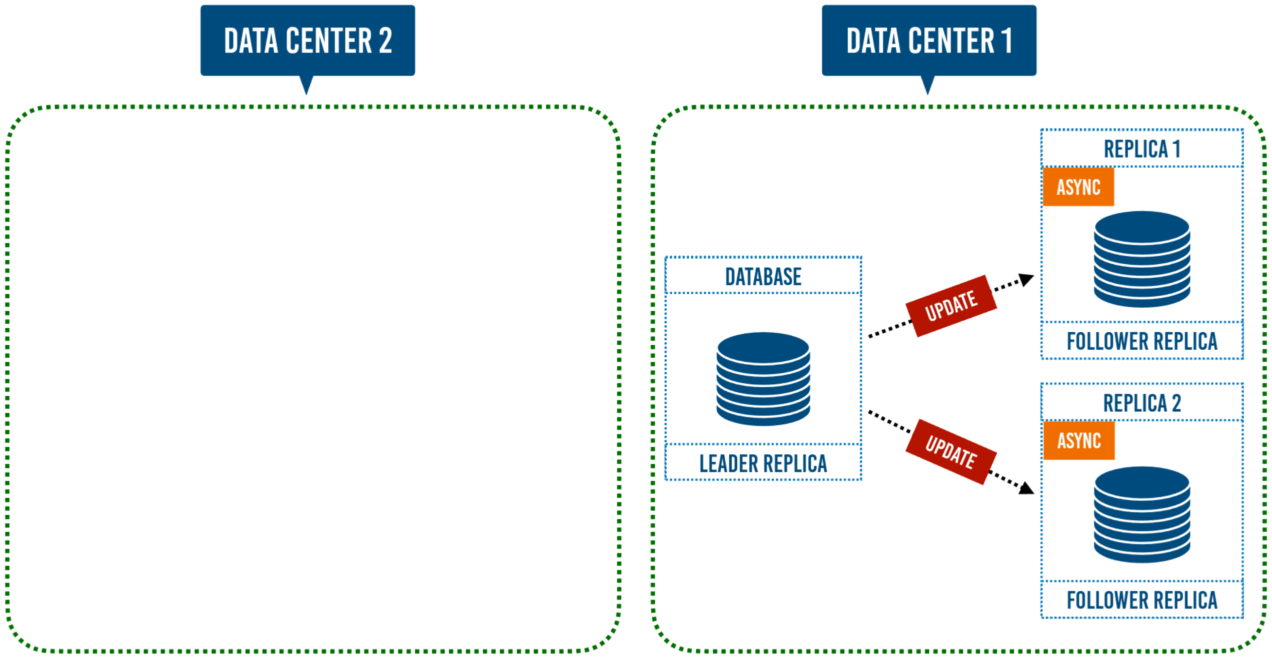
LEADERER-BASED REPLICATION

- **Explanation:** Classic picture of leader-based replication: All writes → leader. Leader sends updates to many followers (usually async). Reads can go to leader or followers → read scaling.
- **Example:** Most web companies (Twitter, Instagram, Shopify…) use leader + many async followers for read-heavy traffic.
- **Key Takeaway:** • Leader-based + async replication = most common production pattern for relational databases today.

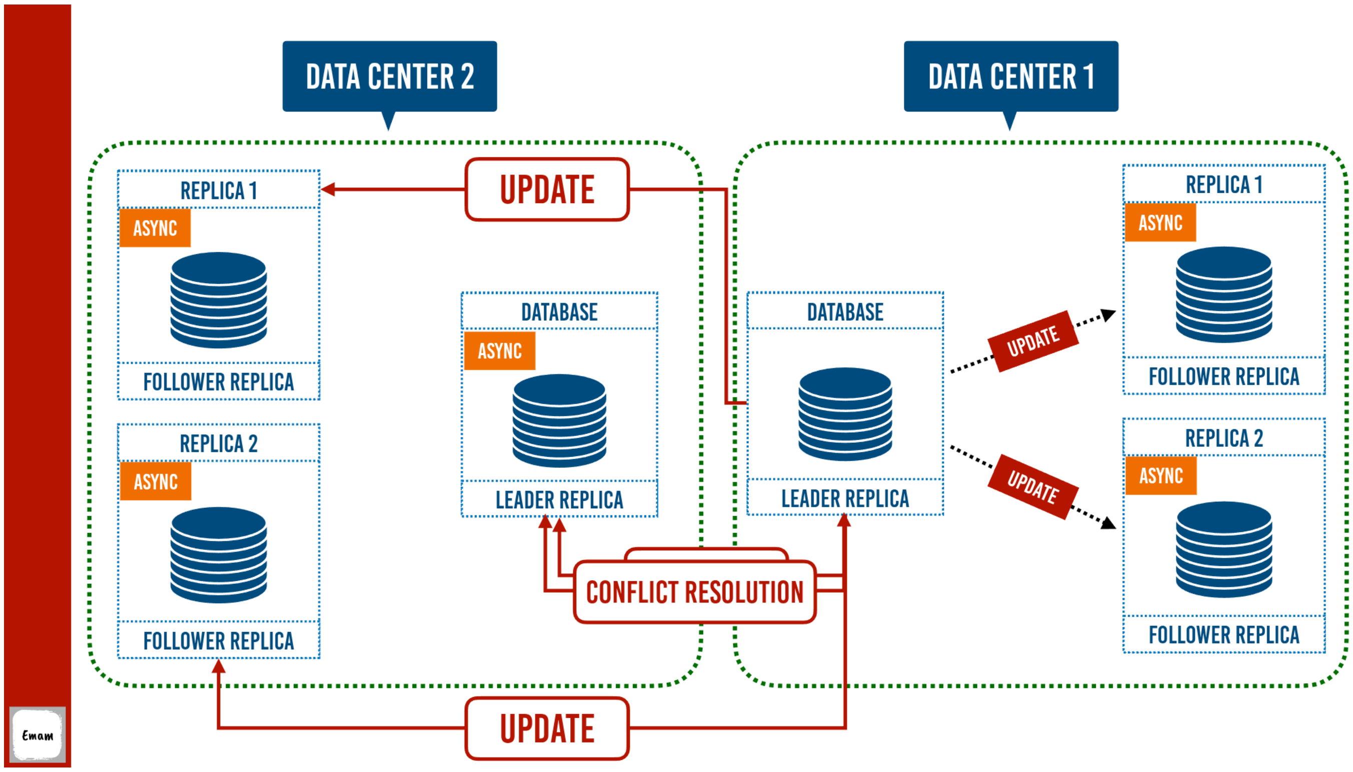
LEADER-BASED REPLICATION



Emam



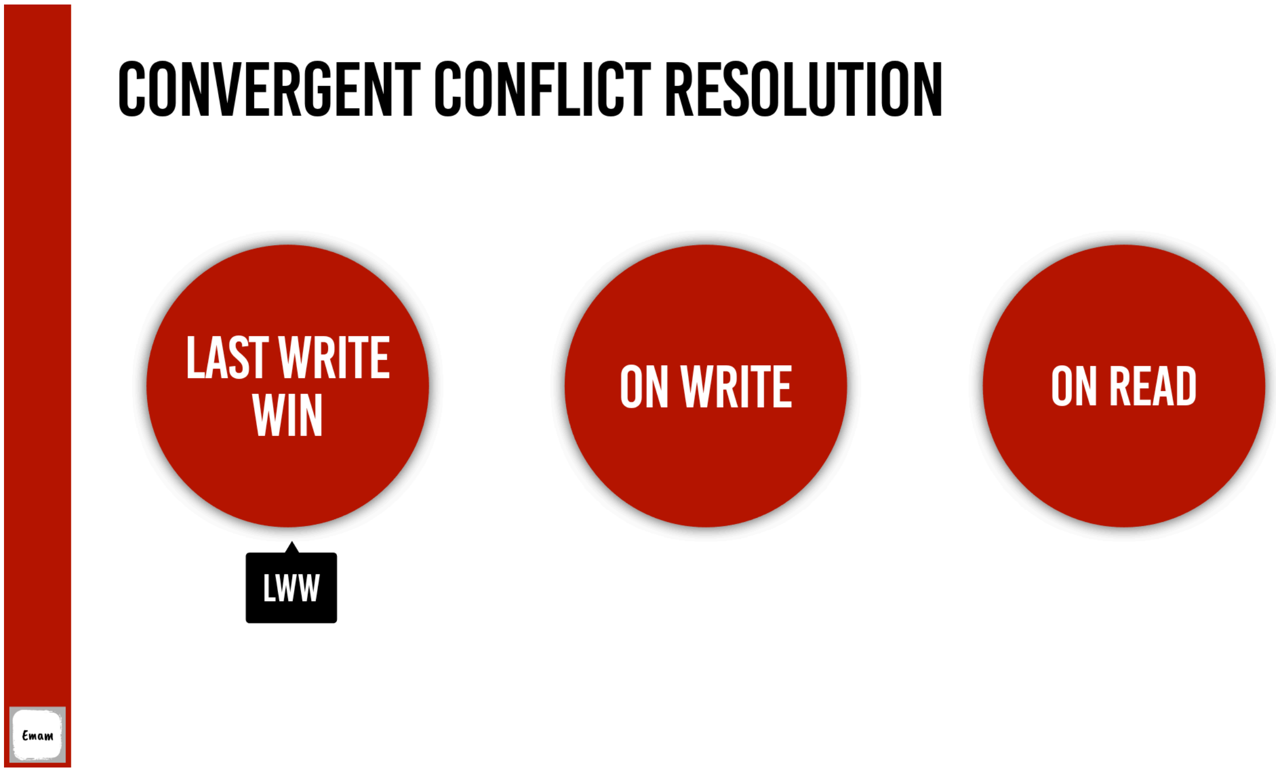
Emam



CONVERGENT CONFLICT RESOLUTION

- **Explanation:** Convergent = all replicas eventually reach the same value. Simplest (but dangerous) way: **Last Write Wins (LWW)** • Give every write a timestamp / unique ID • Keep only the one with highest number/ID → discard others
- **Example:** Cassandra default conflict resolution = LWW (very common). Many mobile apps also use LWW for simplicity.
- **Key Takeaway:** • LWW is easy but loses data silently — only use it when you can accept data loss (caches, non-critical counters…).

I covered the main distinct concepts/slides. If you want deeper explanation on any particular slide (e.g. OT algorithm steps, version vectors math, split-brain prevention techniques…), just tell me which one.



CONVERGENT CONFLICT RESOLUTION

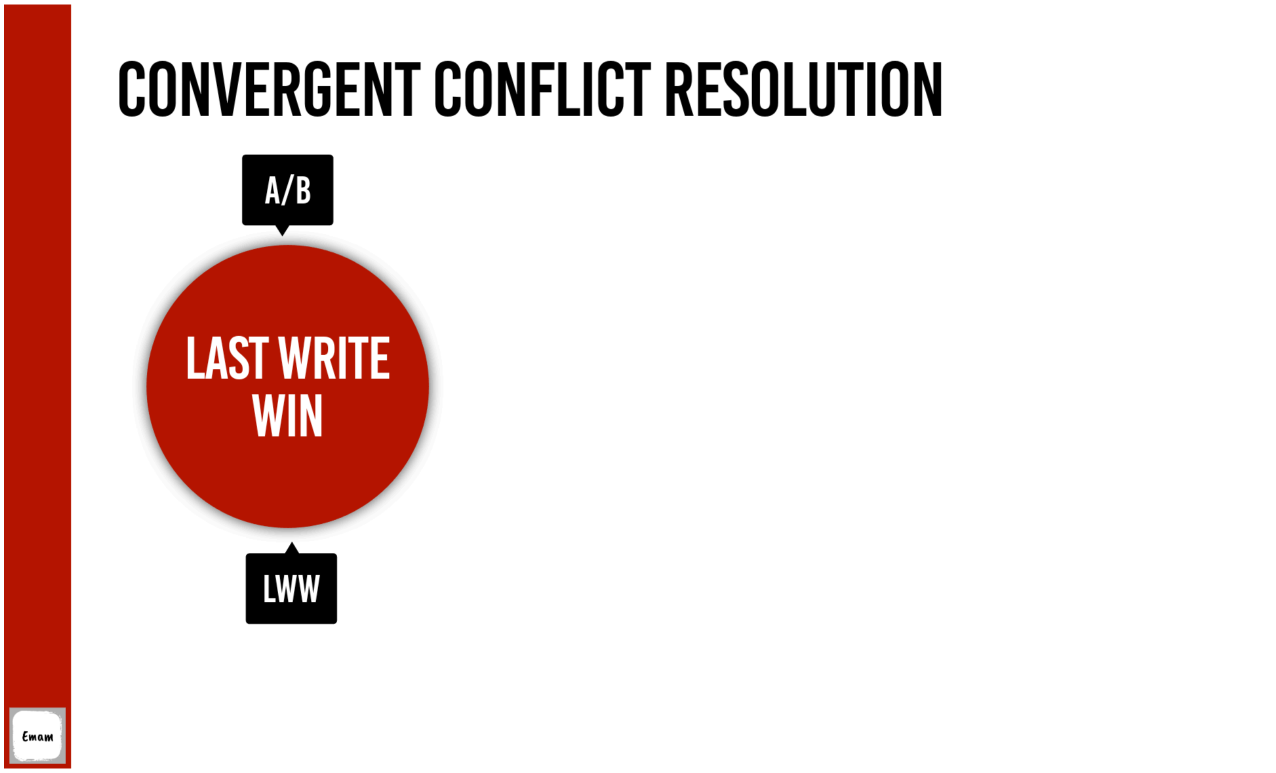
LAST WRITE WIN

ON WRITE

ON READ

LWW

Emam



CONVERGENT CONFLICT RESOLUTION

A/B

LAST WRITE WIN

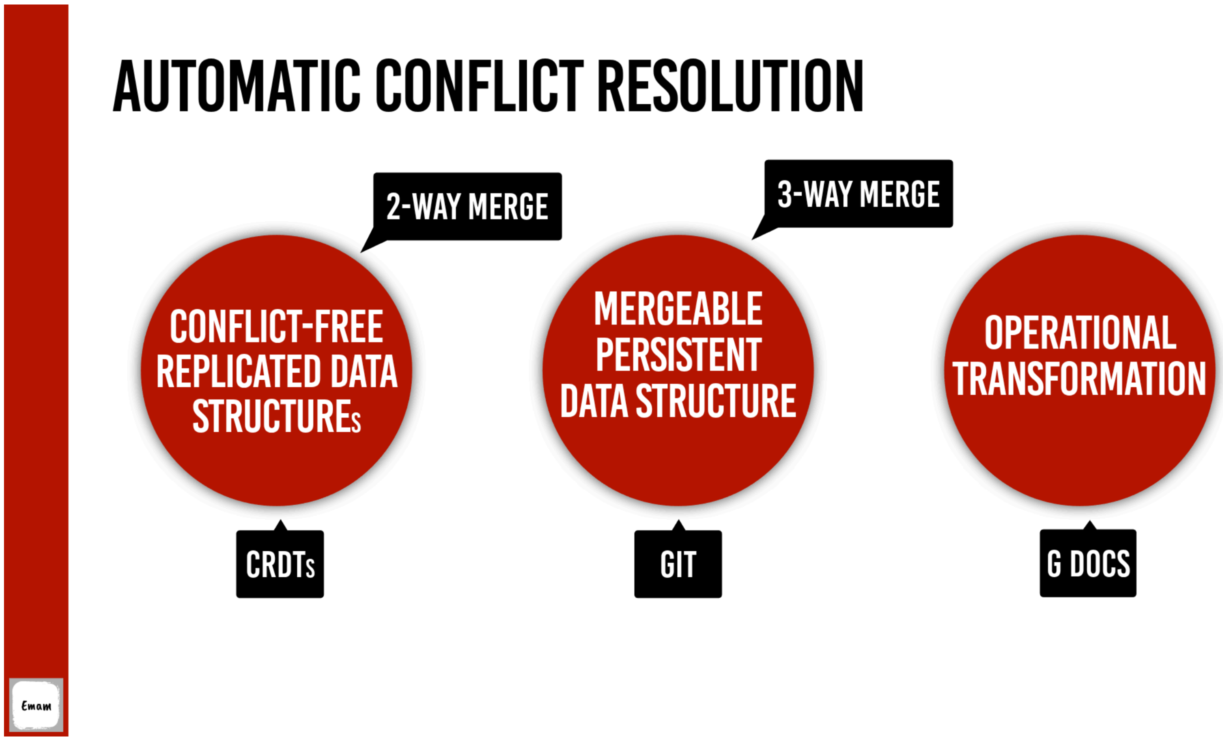
LWW

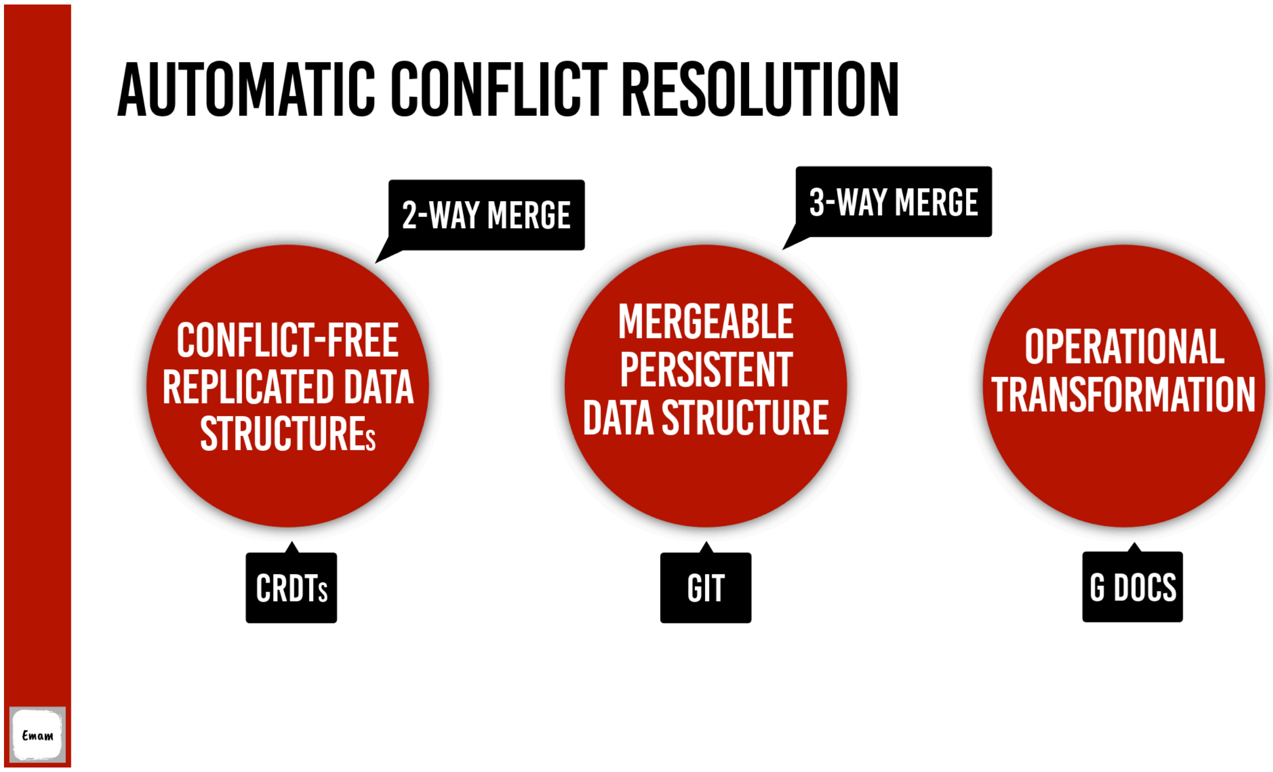
Emam

AUTOMATIC CONFLICT RESOLUTION

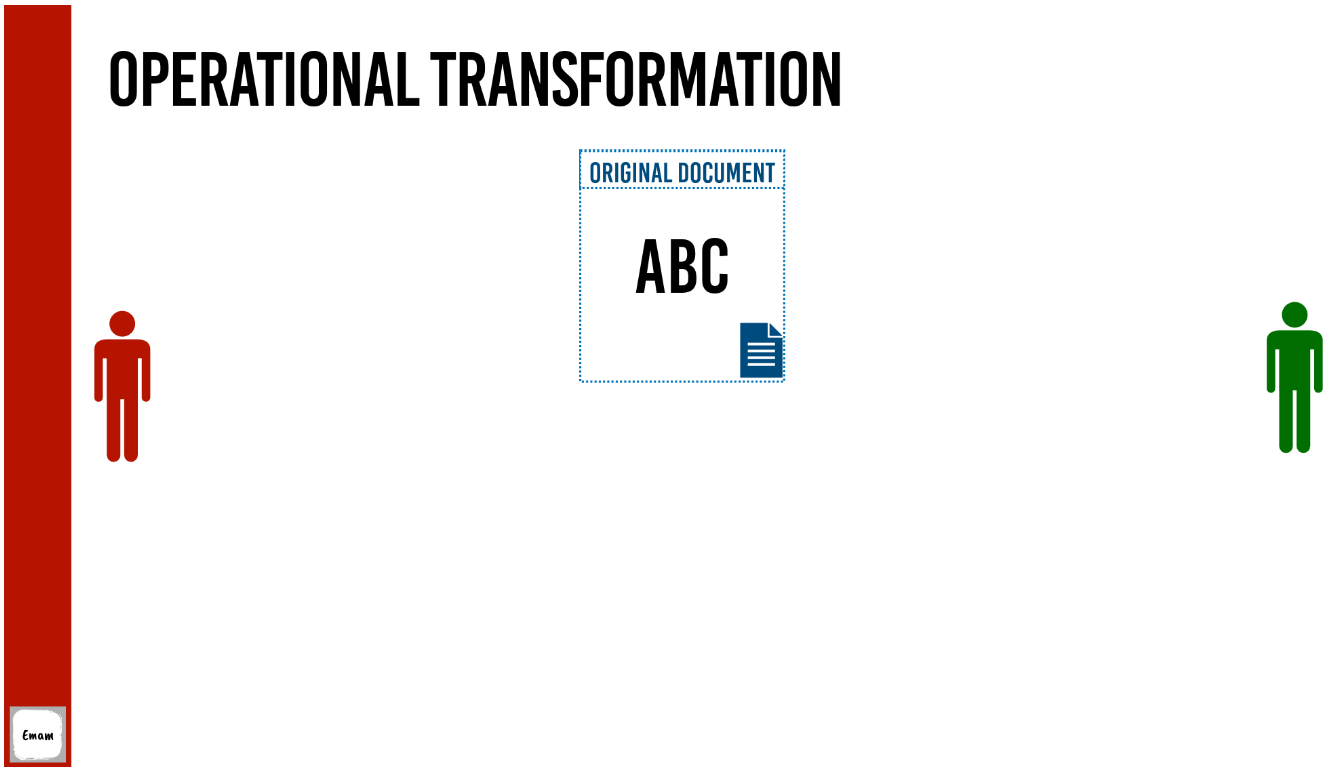
- **Explanation:** When the same data is changed in multiple places at once → conflict. Automatic resolution approaches: • **CRDTs** — special data types that always merge correctly (counters, sets, lists…) • **Operational Transformation** (Google Docs style) • **3-way merge** (like Git) • **2-way merge** (simpler)

- **Example:** Riak → has CRDT counters/sets/maps. Git → 3-way merge for code conflicts. Google Docs → OT for text.
- **Key Takeaway:** • If you want multi-leader or offline-first without manual conflict handling → CRDTs or OT are the modern, clean answers.


AUTOMATIC CONFLICT RESOLUTION
2-WAY MERGE
3-WAY MERGE
CONFLICT-FREE REPLICATED DATA STRUCTUREs
MERGEABLE PERSISTENT DATA STRUCTURE
OPERATIONAL TRANSFORMATION
CRDTs
GIT
G DOCS
Emam



AUTOMATIC CONFLICT RESOLUTION
2-WAY MERGE
3-WAY MERGE
CONFLICT-FREE REPLICATED DATA STRUCTUREs
MERGEABLE PERSISTENT DATA STRUCTURE
OPERATIONAL TRANSFORMATION
CRDTs
GIT
G DOCS
Emam

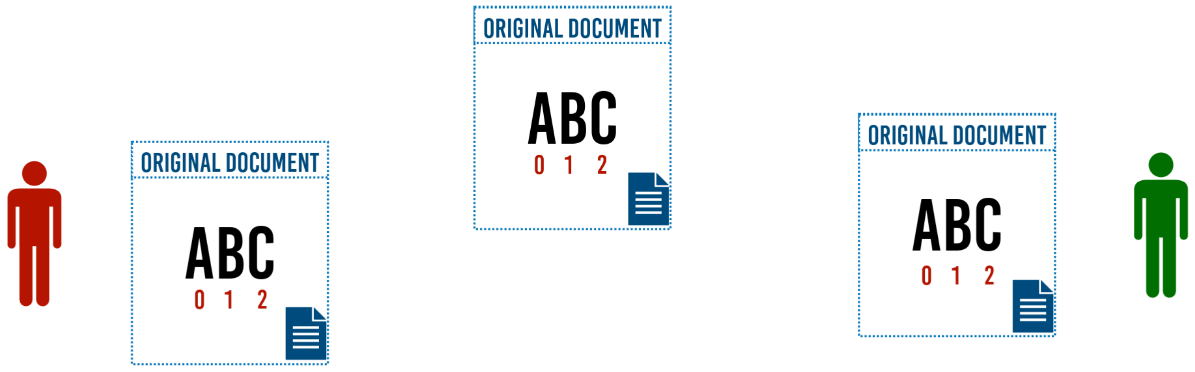


OPERATIONAL TRANSFORMATION

OPERATIONAL TRANSFORMATION

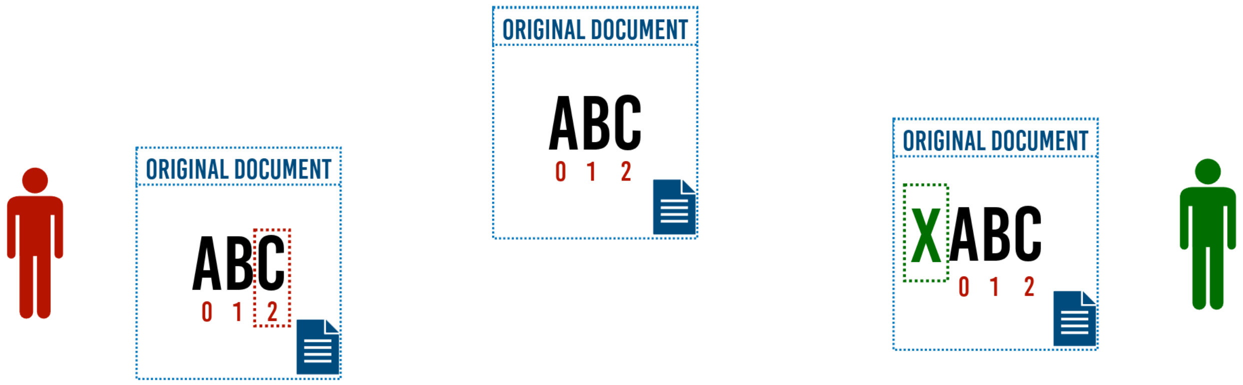
- **Explanation:** Operational Transformation (OT) is a technique used when multiple people edit the **same document at the same time** (Google Docs, Etherpad…). Instead of sending whole document, you send small operations (insert char X at position 5, delete char at 7…). When operations arrive in different order on different clients → OT algorithm **transforms** them so everyone ends up with the same final text (even though operations were applied in different sequence).
- **Example:** Google Docs: you and your colleague type at the same time → your cursor doesn't jump around wildly — OT transforms your colleague's insert so it appears correctly relative to what you just typed.
- **Key Takeaway:** • OT is the magic behind real-time collaborative editing — it turns concurrent operations into consistent document state without locking.

OPERATIONAL TRANSFORMATION



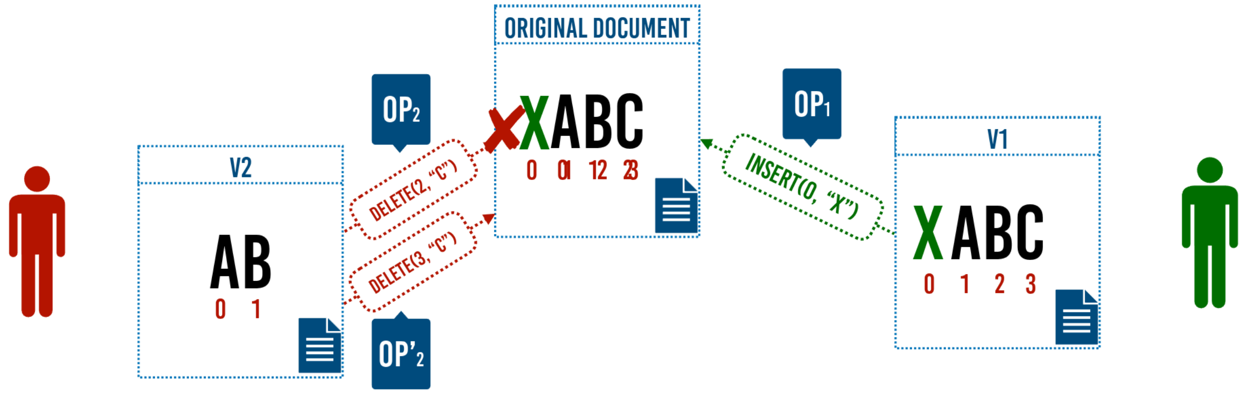
Emam

OPERATIONAL TRANSFORMATION



Emam

OPERATIONAL TRANSFORMATION



ORIGINAL DOCUMENT
XABC
0 0 1 1 2 2 3
OP2
DELETE(2, "C")
DELETE(3, "C")
OP'2
V2
AB
0 1
OP1
INSERT(0, "X")
V1
XABC
0 1 2 3