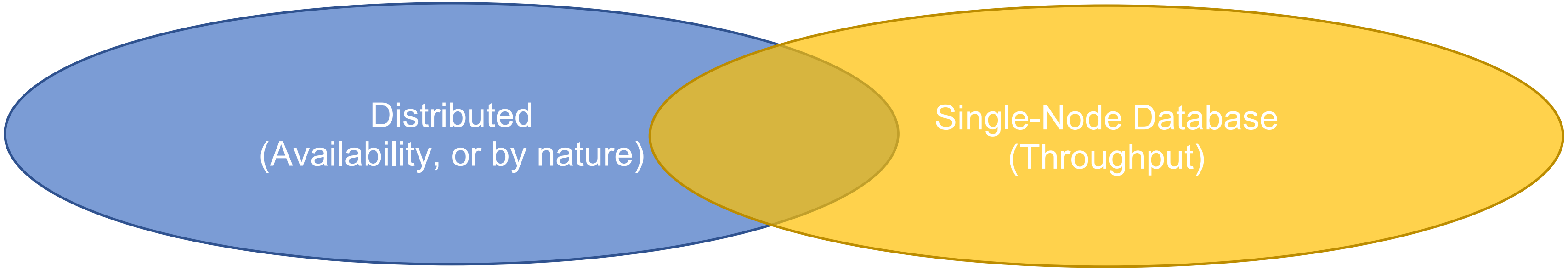
Distributed Systems

Distributed Transactions

CUHK

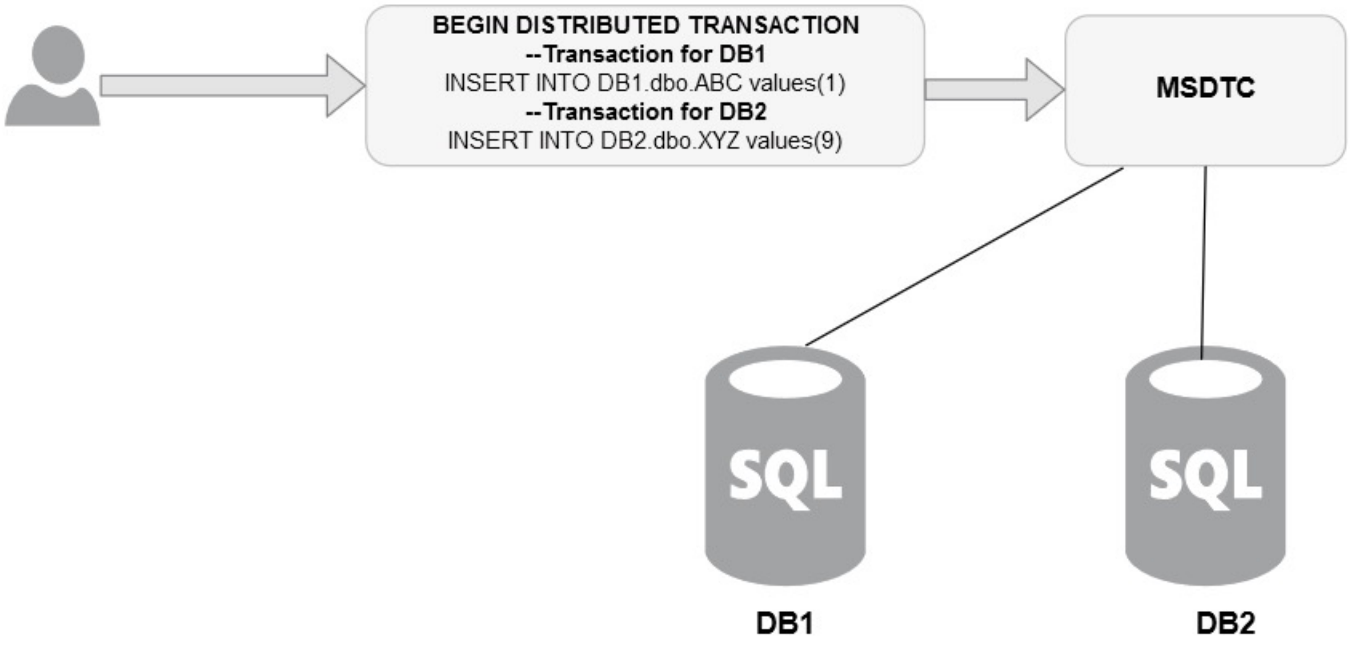
Distributed Databases



Transactions and ACID in (single-node) DB

- T
  - Withdraw $100 from Account A
  - Deposit $100 to Account B
- **A**tomicity: All or Nothing
- **C**onsistency: C in single-node DB != C in distributed world
  - End Result = preserve the invariants (e.g., constraint: account >=0)
- **I**solation: Multiple Concurrent Transactions T1 and T2
  - End Result = **Serializable** = T1 T2 or T2 T1 but not something else
  - Concurrency Control (e.g., locking)
- **D**urability: Result of a committed transaction persist no matter what
  - e.g., earthquake, machine failure
  - Logging and Recovery (not our focus)

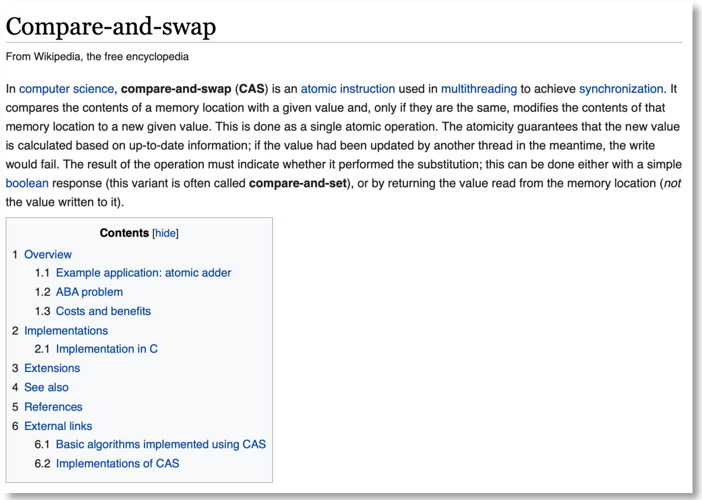
Distributed Transactions and ACID



https://www.sqlshack.com/explore-cross-database-msdtc-for-distributed-transactions-in-sql-server-always-on-availability-groups/

Concurrency control (OS vs DB) on single node

- OS:
  - Concurrent update to a **single-value**
    - Locking (Semaphore; Pthread lock)
    - Lock-free (Use of CAS)
- DB:
  - A **transaction** is a bigger unit
    - T involves R/W on **multiple values with ACID**



Compare-and-swap

From Wikipedia, the free encyclopedia

In computer science, **compare-and-swap** (**CAS**) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value and, only if they are the same, modifies the contents of that memory location to a new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple boolean response (this variant is often called **compare-and-set**), or by returning the value read from the memory location (*not* the value written to it).

Contents [hide]

AC*I*D on single-node DB

- Pessimistic (~ to OS Lock-based)
    - 2 Phase Locking (2PL)
        - As long as a txn wants to access (read/write) an item, it must acquire a lock of that item first
        - A txn that touches multiple items
            - once unlock //then step into phase 2
                - **you can't acquire any lock again**
        - 2PL guarantees you a **serializable** schedule = the order of the final lock acquired
        - Good: little coordination between transactions
            - Each transaction locally respects 2PL protocol and access the lock table
        - Bad: deadlock, lock overhead
    - Fine-grained locks: reader-writer
- Optimistic (~ to OS Lock-free)
    - Instead of waiting for the lock
        - let them interleave whatever, but abort on observing conflicts and restart

	S	X
S	✔	✗
X	✗	✗

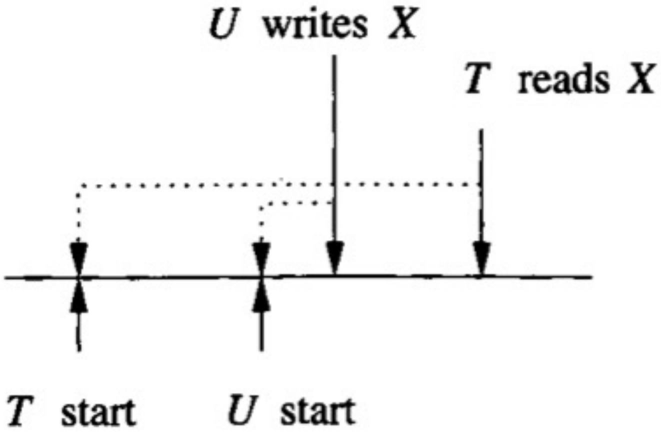
Ref: geeks4geeks

ACID on single-node DB

- Optimistic
  - Timestamp-ordering based
    - Cross check between timestamps of txns on data items **during read and write**
    - The equivalent serializable order = timestamp order of the transaction
    - E.g.,
      - Serializable order is: T < U
      - Whenever you (T) **read** X
        - Last-time-X-**written**-by-others (e.g., U)

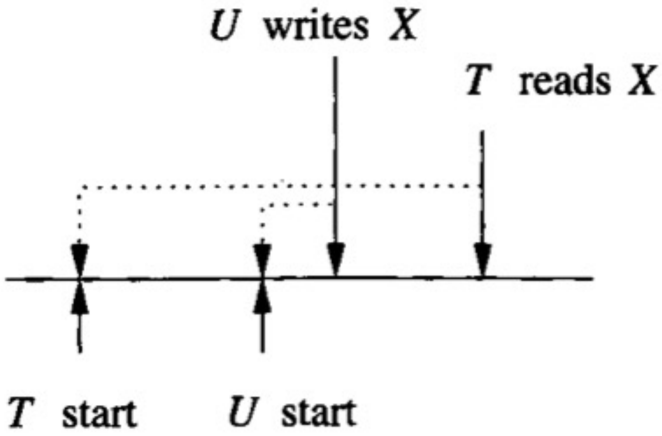
          >

          Your start timestamp (i.e., T start)
        - "Read too late" ➔ Abort and Restart T



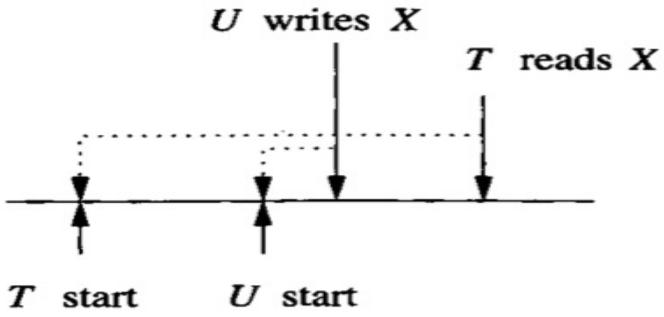
AC/D on single-node DB

- Optimistic
  - Timestamp + Multi-versioning
    - If keeping versions of X
      - Then T can read the previous version of X



ACID on single-node DB

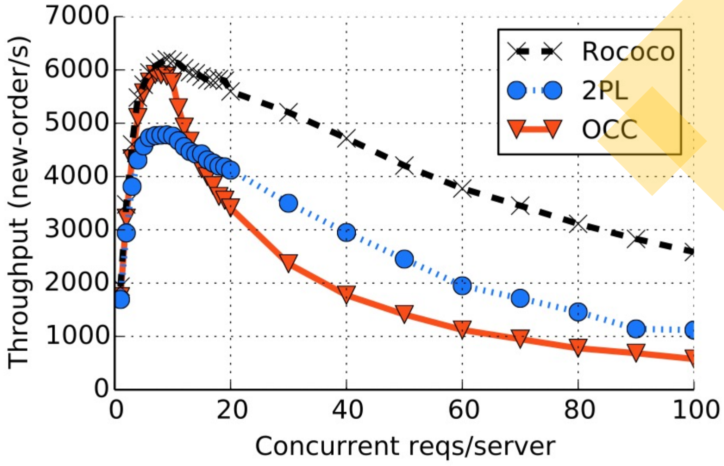
- Optimistic
  - Validation based (a.k.a. OCC*)
    - Space overhead spent on **active transactions only**
  - Read Phase 1:
    - The writes are written into **thread-local in-memory write set** (not to DB yet)
  - When a transaction T is ending
    - Validation Phase 2:
      - Validate its read-set and write-set of T overlap with other active transactions' r/w-sets?
    - If no overlap / no serializability conflict / no problem after reordering
      - Apply the write-set //write Phase 3
    - Else
      - Abort T and Restart T



*very confusing naming convention; But we follow Complete book convention here

Pessimistic vs Optimistic

- Locking
  - Setting a lock on/off = ***writing shared memory*** 0/1 = overhead
  - If low-contention
    - (e.g., T1 writes X but T2 reads Y)
    - Locking overhead is for nothing
    - → hurt throughput
- Optimistic
  - If low-contention
    - → little overhead → better throughput
  - If high-contention
    - → Abort&Restart&Abort&Restart.. → waste of work



(a) Throughput

Transaction is the holy grail in systems research

- Optimistic Lock: 64-bit Lock = <version + lock-bit>
  - Put(k): //lock-based + version

```
Write-lock(k) {
    if lock-bit=1
        restart
    set lock bit=1 (exclude other writers)  }
Write k into memory table;
Write-unlock(k) {
  set version++, set lock-bit =0} //put lock and version in the same word
```

  - Get(k):

```
restart:
vers = Read-lock(k){
        If lock=1 → restart   //No shared memory write here ☺
        Else return version}
Read k from memory table;
success = Read-unlock(vers) {
    If version unmatch or lock is on}
If !success ➔ goto restart
```

Notion of Correctness under Concurrency

- Linearizability (Shared Memory, OS):
  - Multiple threads/processes invoke operations on a single object (e.g., linked list)
    - But it must behave like operations happen one-by-one (serial) following **real-time order**
- Serializability (Shared Memory, Single-Node DB):
  - Multiple threads (transactions) on multiple items
    - But it must behave like **any** serial order
      - So: T1 T2 T3 or T2 T1 T3 is also fine
- Linearizability (Replicated State Machine, Distributed):
  - Multiple replicas on a "single" logical object (e.g., a log, a kv store)
    - But behaves like a single-threaded machine

http://www.bailis.org/blog/linearizability-versus-serializability/

ACID on distributed DB

- Problem: any site may fail or disconnect while a commit for transaction T is in progress.
    - Atomicity says
        - T does not “partly commit”, i.e., commit at some site but abort at the others.
        - Individual sites cannot unilaterally choose to abort T without the agreement of the other sites.
        - **If T holds locks at a site S,**
            - **then S cannot release them until it knows if T is committed or aborted.**
        - If T has pending updates to data at a site S,
            - then S cannot expose the data until T commits/aborts.

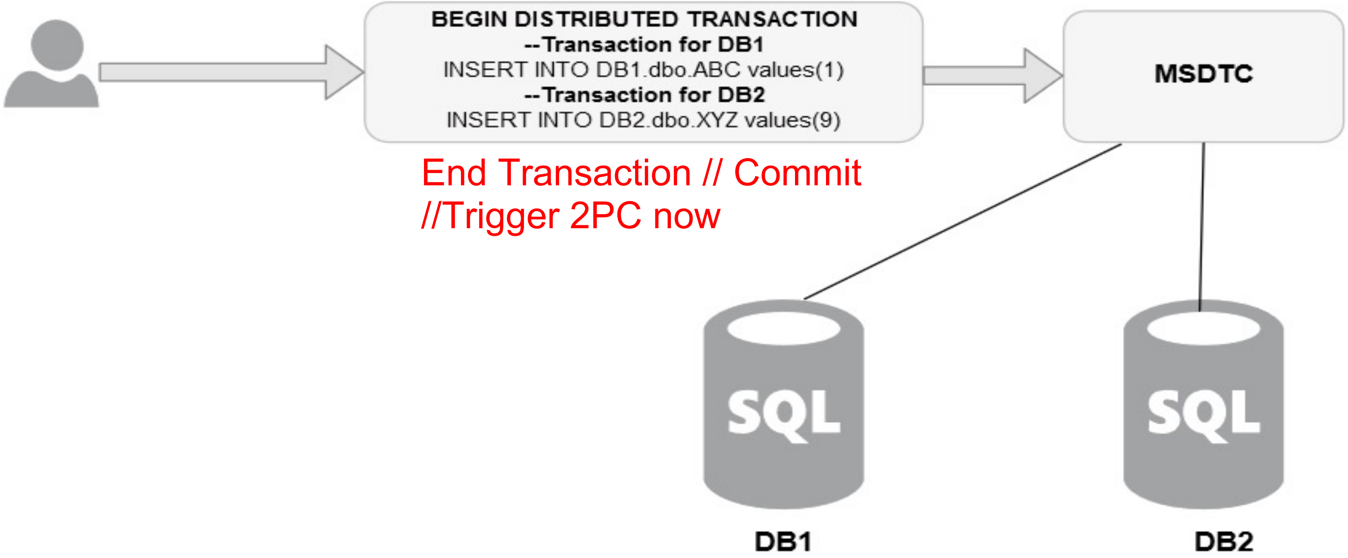
ACID on distributed DB

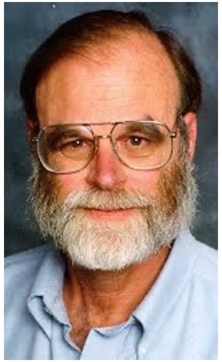
STEVE SILBERMAN BUSINESS 07.24.2007 12:00 PM

Inside the High Tech Hunt for a Missing Silicon Valley Legend

The news that Gray was missing shocked the high tech community. The lanky coder had been a computing legend since the 1970s. His work helped make possible such mainstays of modern life as cash machines, ecommerce, online ticketing, and deep databases like Google.

- 2PC
  - Needs a **designated** coordinator (so if that is dead, hanged); vs. Raft: any one can be a leader





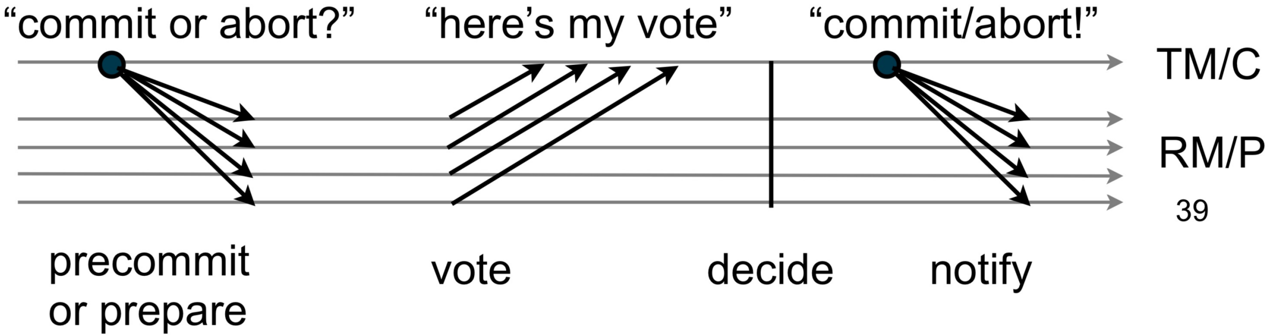
https://www.sqlshack.com/explore-cross-database-msdtc-for-distributed-transactions-in-sql-server-always-on-availability-groups/

2PC

- TM = tran manager (coordinator)
- RM = remote manager (Participant)

If unanimous to commit
decide to commit
else decide to abort

Might block forever if the coordinator (a.k.a. transaction manager) fails or disconnects



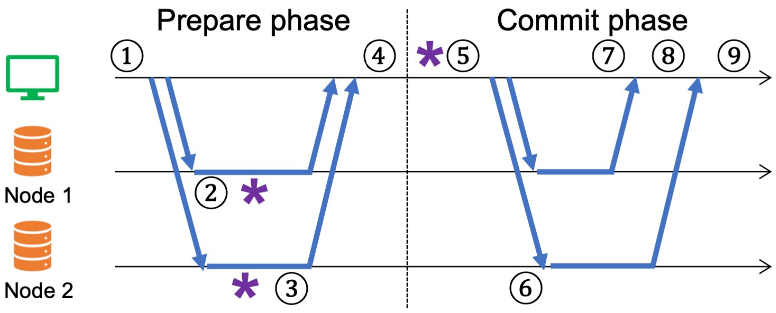
RMs validate Tx and prepare by logging their local updates and decisions

TM logs commit/abort (commit point)

29

https://www2.cs.duke.edu/courses/fall07/cps212/consensus.pdf

9 cases of fail-stop during 2PC



(1) before the coordinator sends prepare requests

(2) after some participant nodes receive prepared requests

(3) after all participant nodes receive prepared requests

(4) before the coordinator receives all votes from participant nodes

(5) after the coordinator writes the commit record

(6) before some participant nodes receive commit requests

(7) before the coordinator receives any acknowledgement

(8) after the coordinator receives some acknowledgements, and

(9) after the coordinator receives all acknowledgements.

Lu et al. VLDB 2021

Availability angle: 2PC is not fault-tolerant

- All processes agree on {Commit, Abort}
- To counter FLP
  - We need safety more than liveness in this $ case
- 2-Phase Commit (centralized)
  - Might block forever if the coordinator (a.k.a. transaction manager) fails or disconnects
  - i.e., Not (coordinator) fault-tolerance (Paxos is exactly fault-tolerance)
  - All sites are voters (unanimous votes vs Paxos: quorum voting)
- 3-Phase Commit (centralized)
  - With a coordinator
  - Add one more phase to achieve fault-tolerance
- Can Paxos solve atomic commit? Yes
  - Paxos Commit [Gray and Lamport; 2x Turing Award; 2004]
    - So, it's also fault-tolerance and **<u>decentralized</u>**
    - Less efficient than 2PC (which is fair because it is more fault-tolerance) and more efficient than 3PC
    - In Distributed Transaction, processes **do have stronger voting right (to say no / abort)**
      - But remember in Paxos? A process is more "whatever choice" and just want to get the consensus done asap

Scalability angle: 2PC is a bottleneck of distributed DB

- Locks must be held until 2PC steps into phase 2
- Significantly hurt concurrency ➔ hurt throughput/latency/scalability
- Motivated NoSQL movement

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

OSDI'06

SQL Databases v. NoSQL Databases

Michael Stonebraker considers several performance arguments in favor of NoSQL databases—and finds them insufficient.

From Michael Stonebraker's "The NoSQL Discussion has Nothing to Do With SQL"

http://cacm.acm.org/blogs/blog-cacm/50678

group identifies itself as advoc
NoSQL.
There are two possible rea
to move to either of these alte
DBMS technologies: performanc
flexibility.
The performance argument
something like the following: I st

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here with our views on MapReduce. T
"cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this sug
number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program su
freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the develo
purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

Consistent vs Availability vs Scalable

	NoSQL (<2007)	NewSQL (>2008)		
Example	Cassandra, DynamoDB, BigTable, …	H-store (2008)	Spanner (2013)	TAPIR, Carousel, CALM, SLOG, … (2013+)
C v A v S	Weaker C for A S	C S over A	Said have SQL demand 😂 Their workloads **can't avoid** cross-shard txn ➔ need 2PC **Txn + SQL + sharding + 2PC + serializable + C over A** + **<u>linearizable(?) RSM</u>** **After all these years, isn't that a distributed SQL database + replication?** 😂	Making CA with better S now Make 2PC+RSM integrate better for higher efficiency; see 2PC+RSM papers Or do away coordination; see SLOG and CALM, etc.
Scalability	KV for S: ⇒ No multiple items **txn** ⇒ No need to consider **serializability; so no CC** ⇒ **No txn -> No cross-shard;** so **<u>no 2PC</u>** ⇒ NoSQL is their (negative) consequence, not the cause	Demonstrated that - **txn + SQL + sharding + serializable can <u>scale</u>** - By *avoiding* cross-shard txn; **so no 2PC**		

NoSQL is a "giant step backward" (2008)