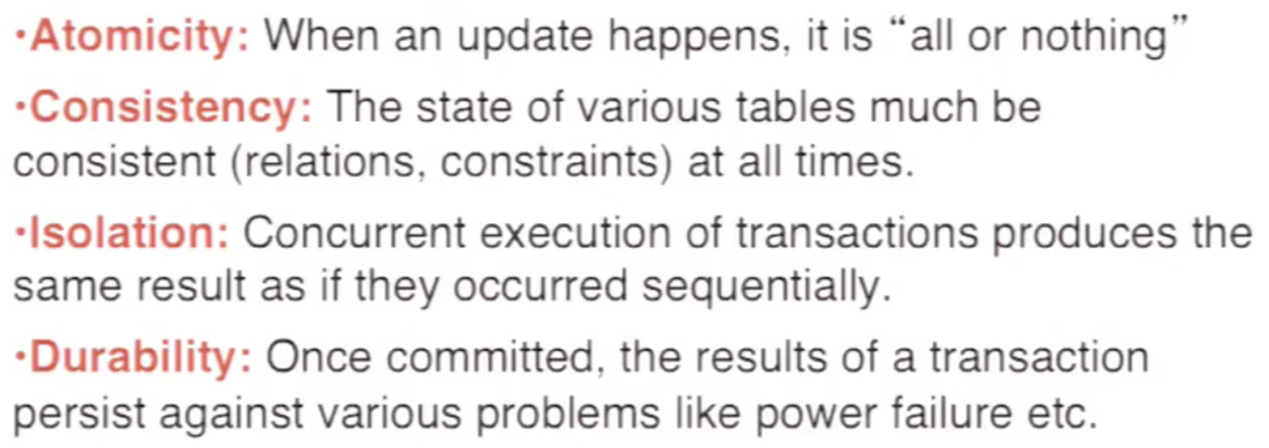
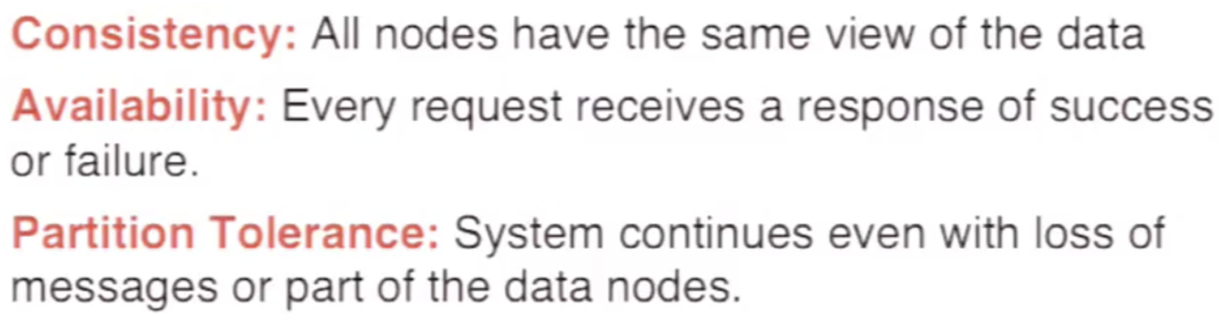
key value storage, databases & transactions

- just has a `put(key, value)` and `get(key)` interface
- often used as a large scale distributed "database". database is in quotes because these often relax some of the consistency guarantees provided by databases
- often distributed over thousands of machines
- traditional databases have ACID properties:



- **Atomicity:** When an update happens, it is "all or nothing"
- **Consistency:** The state of various tables much be consistent (relations, constraints) at all times.
- **Isolation:** Concurrent execution of transactions produces the same result as if they occurred sequentially.
- **Durability:** Once committed, the results of a transaction persist against various problems like power failure etc.

- CAP theorem: in a distributed system, we can't get all three of the following



Consistency: All nodes have the same view of the data

Availability: Every request receives a response of success or failure.

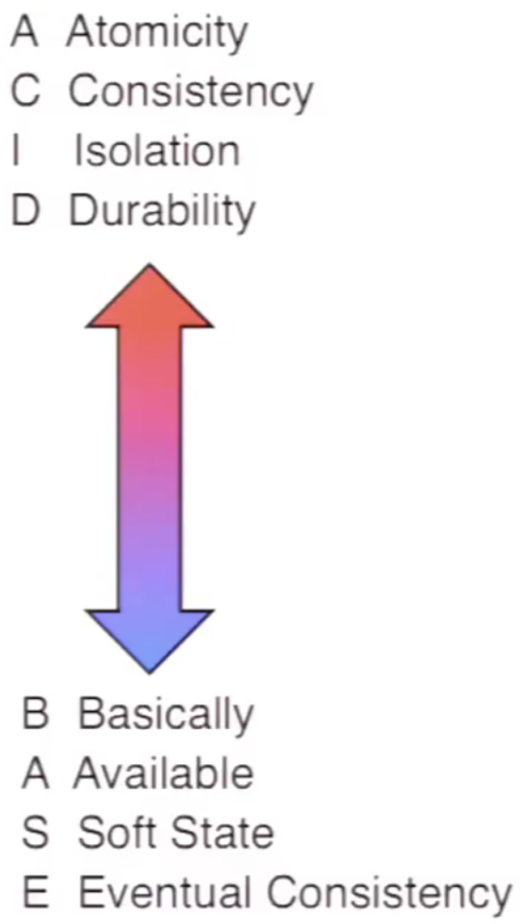
Partition Tolerance: System continues even with loss of messages or part of the data nodes.

- a database can be split into key value stores by making one of the columns the key and each row can be split into multiple key value pairs with whatever you need the index being the key
- the challenges with a distributed database are:
  - fault tolerance → machines will fail at a large scale
  - scalability → gotta have reasonable read write times at scale
  - consistency → ideally we'd like all nodes to provide a good, consistent interface
- a recursive query is one where you go to the master, and it finds the data and gets it back to you. in an iterative query, it tell you where to go and you go there. recursive could be quicker because there aren't as many hops across the internet (hops are within datacenter), but ofcourse master is now the bottleneck.
- replicating data for machine failures introduces challenges of consensus in case of disagreement.
- another idea is to replicate more the hot keys. if its a read heavy workload, this gets a bit simpler.

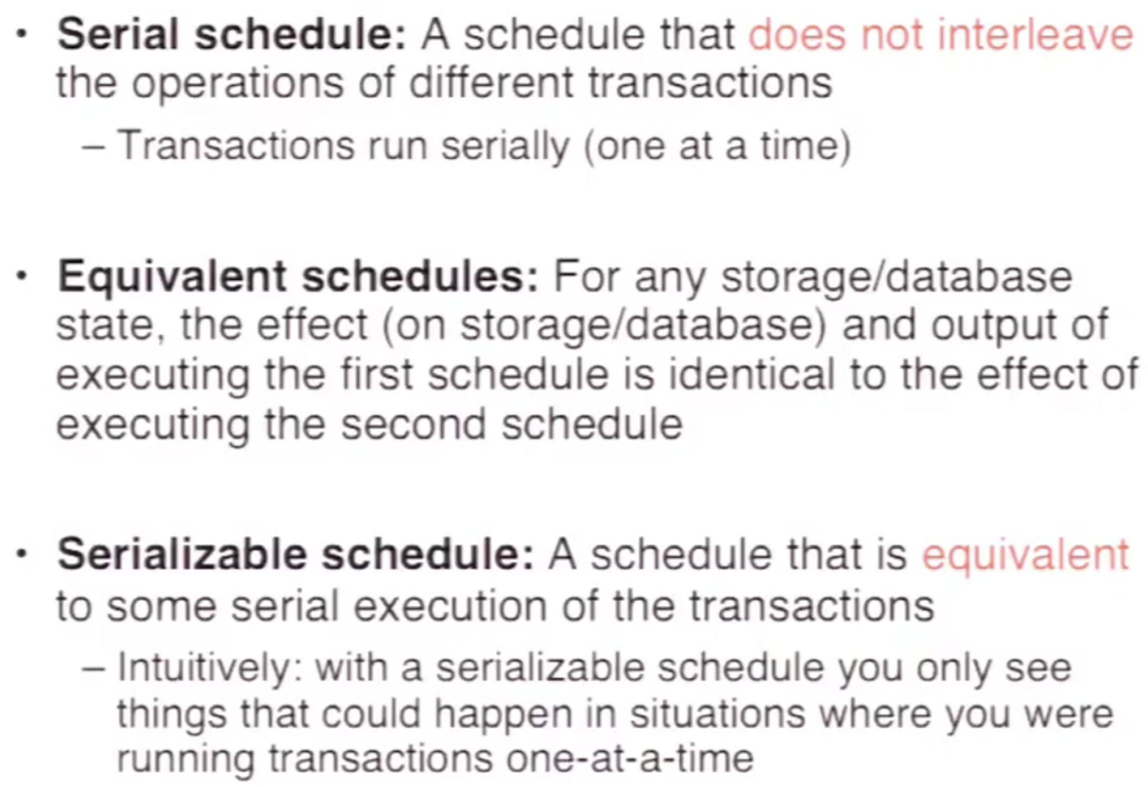
- quorom consensus: if we have a replica set of size N (this means that there are N replicas of each data), lets say when we write we will wait for responses from W replicas and when we read we will wait for responses from R replicas, if W + R > N, then by pigeonhole principle, at least one data node who ack'ed the write has submitted its response to the read request.
- You can trade off W vs R depending on if you want better read performance or better write performance.

databases & transactions

- when choosing a DB, the most important question is what kinds of properties are desirable from the database? consistency or fast reads and writes with high availability and scalability?
- a relational DB has entities and relationships
- A **transaction** is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID)
- we can have multiple read locks but just one write lock. why? because we can have multiple requests reading the same data but we want writes to be atomic and isolated.
- if we are updating many tables, we might want to lock the whole table. another level of granularity is to lock at the table level. ideally we just lock at the row level. more granularity → more concurrency
- The opposite of ACID systems are those that provide very high availability and eventual consistency but less on consistency and isolation (ACID vs BASE)

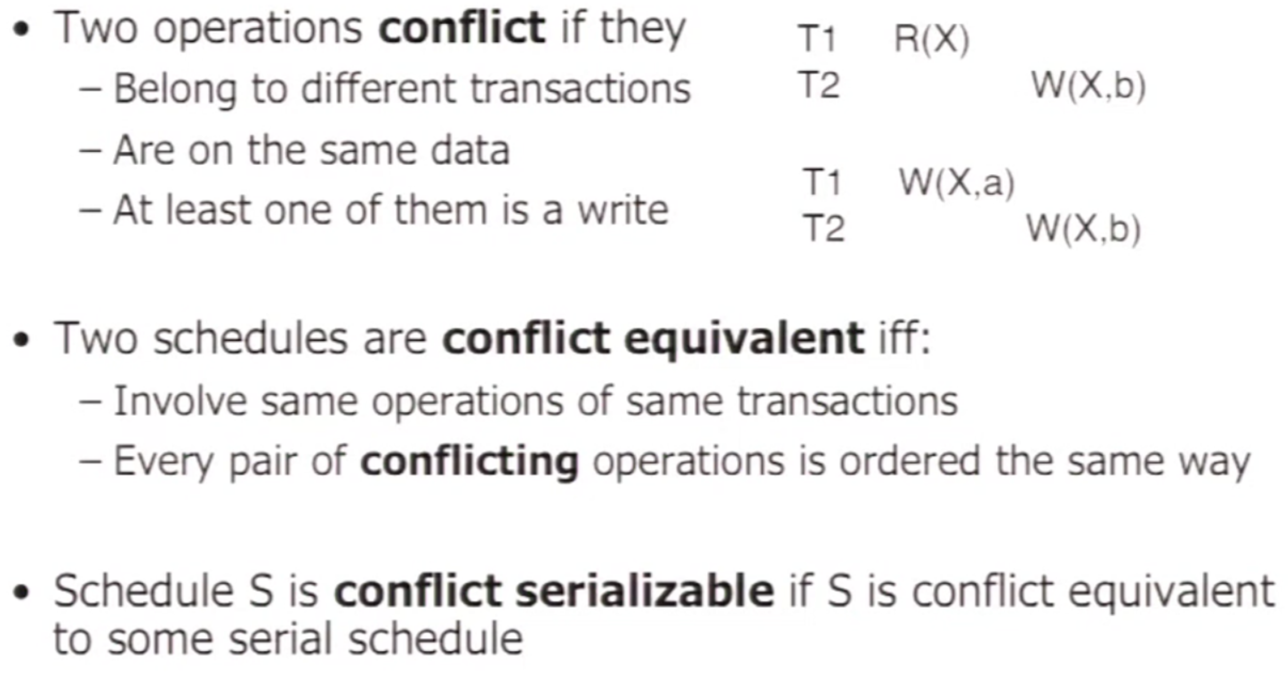


- terminology regarding transaction scheduling



- **Serial schedule:** A schedule that does not interleave the operations of different transactions
  - – Transactions run serially (one at a time)
- **Equivalent schedules:** For any storage/database state, the effect (on storage/database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions
  - – Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

- transactions executed totally one after the other obviously follow a serial schedule and hence are correct. but, what we'd like to do is to find a serializable schedule that's more efficient than executing them one after the other
- a conflict occurs when one or more operations in different transactions are operating on the same data and at least one of them is a write. some more concepts



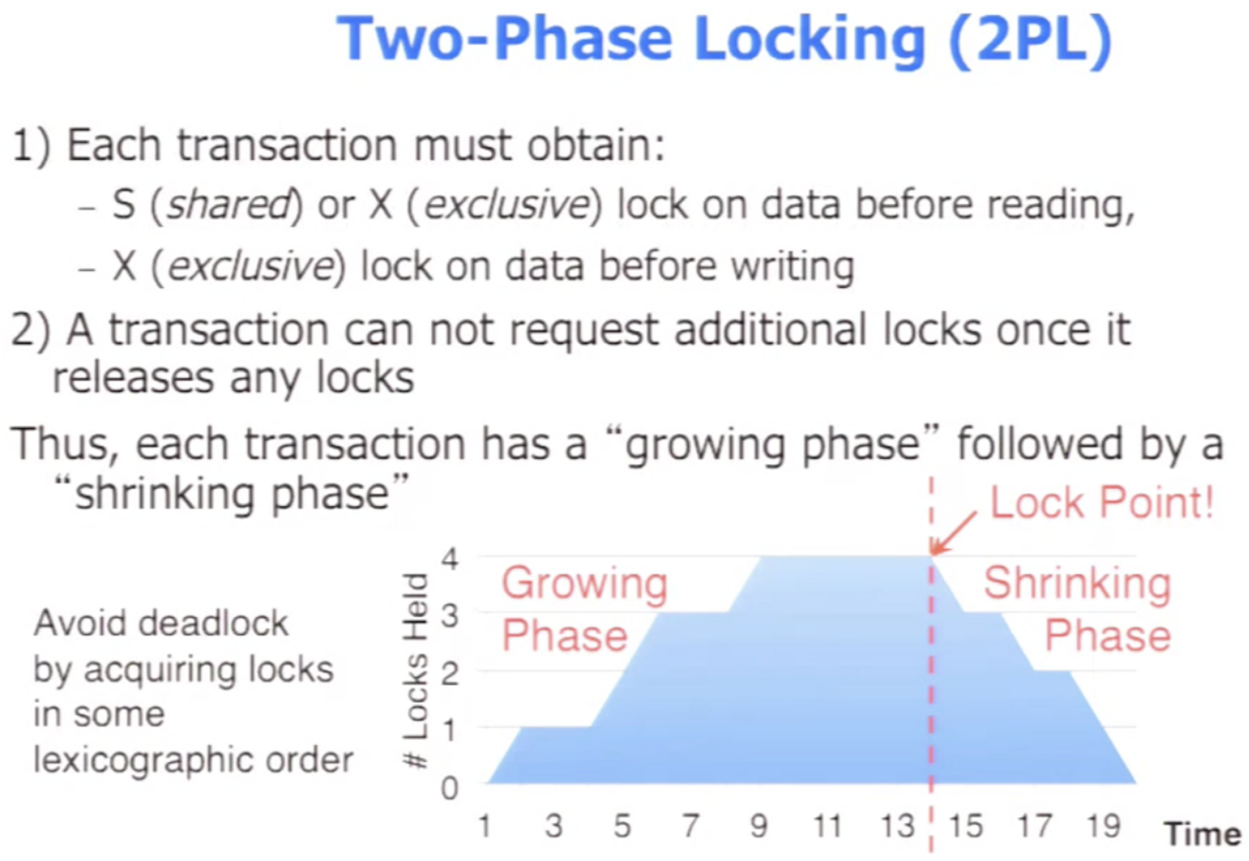
- Two operations **conflict** if they
  - – Belong to different transactions
  - – Are on the same data
  - – At least one of them is a write

```
T1   R(X)
T2              W(X,b)

T1   W(X,a)
T2              W(X,b)
```

- Two schedules are **conflict equivalent** iff:
  - – Involve same operations of same transactions
  - – Every pair of **conflicting** operations is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

- Ideally given a set of transactions, we want to find a conflict serializable schedule that is efficient so that our ACID database can also be performant.
- 2PL (2 phase locking) is a way to implement conflict serializability



Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - S (*shared*) or X (*exclusive*) lock on data before reading,
   - X (*exclusive*) lock on data before writing

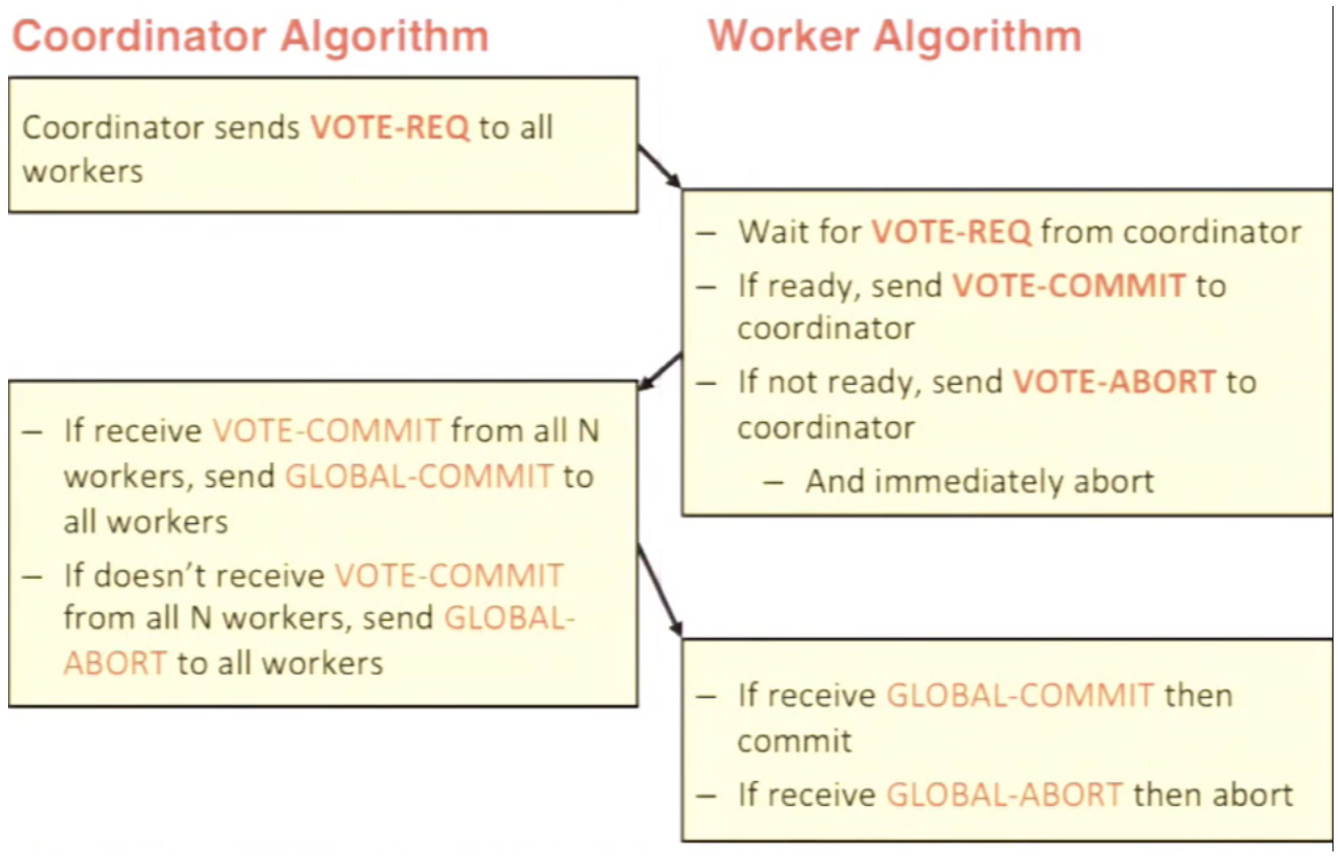
2) A transaction can not request additional locks once it releases any locks

Thus, each transaction has a "growing phase" followed by a "shrinking phase"

Avoid deadlock by acquiring locks in some lexicographic order

Lock Point!
Growing Phase
Shrinking Phase
# Locks Held
0
1
2
3
4
1
3
5
7
9
11
13
15
17
19
Time

- strict 2PL is when you hold onto all acquired locks till the end and after you're done with all work, release them all at once
- There are deadlock detection strategies like timeouts and cycle detection that can be used by a lock manager for detecting and unblocking deadlocks. another strategy is to claim locks in lexicographic order.
- once a distributed transaction is done, if we want to commit a transaction in all replicas, we use something called a 2 phase commit (2PC) algorithm:



Coordinator Algorithm

Worker Algorithm

Coordinator sends **VOTE-REQ** to all workers

- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send **VOTE-ABORT** to coordinator
  - And immediately abort

- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort