In reality, a lot can go wrong in a data system, and building fault-tolerant systems is difficult because it is almost impossible to reason through every possible error.
In response to this, transactions have been the mechanism of choice to simplify these issues.

A transaction is a way for an application to group several reads and writes together into a logical unit -- so all reads and writes in a single txn are executed as one operation -- either the entire thing succeeds (commits) or it fails (rollback, abort),
- it it fails, the app can safely retry -- the application does not need to worry about *partial* failures
- the transaction is not a law of nature -- it is a human created abstraction to simplofy the programing model for applications accessing a database. The appication can safely ignore some potential error scenarios and concurrency issues

The meaning of ACID

Atomicity
Consistency
Isolation
Durability

Atomicity

In general, atomic refers to something that cannot be broken down further into smaller parts.

- in multi-threaded programming, if one thread executres an atomic operation, that means there is no way another thread could see the half-finished result of the operation. The system can either be in the state it was before the operation or after
- however, in the context of ACID, atomicity is not about concurrency -- this is covered by I, isolation -- which describes what happens when several processes try to access the same data at the same time
- here, it describes what hapens if a client wants to make several writes but a fault occurs after some of the writes have been processed -- for example, a process crashes, a network connection is interrupted, etc
- if the wrties are group together into an atomic transaction, and the txn cant be completed due to a fult, then the txn is aborted and the db must discard or undo any writes it has made so far in that txn
- without atomicity, if an error occurs partway through making multiple changes, it's difficult to know which changes have taken effect and which haven't
    - the application can try again but that risks making the same change twice -- leading ot duplicate or incorrect data
- the ability to abort a trxn on error and have all writes from the transaction discarded is the defining feature of ACID atomicity

    - think of it as *abortability*

Consistency

- another terribly overloaded term
    - replica consistency and eventual consistency?
    - consistent hashing?
    - consistency as linearizability as in CAP theorem?
- in ACID, the idea is that certain invariants about the data must always be true -- for example, credits and debits must be balanced across accoutns
- this is typically a property of the *application* and not the db -- the db is not inherently aware of business logic/rules of this form
    - the app may rely on the A and I properties to achieve C but it's not inherent to the db

Isolation

- in ACID, this means that concurrently executing transactions are isolated from each other -- they cannot step on each other's toes
    - for example, if a txn makes several writes, then another txn should see either all or none of those writes, but not some subset
    - this is often formalized as serializability: each transaction can pretend it is the only transaction running on the entire db
    - the db ensures that when the txns have commited, the result is the same as if they had run serially (one after another), even though they may have run concurrently in reality
    - in practice this carries a performance penalty
    - Oracle substitutes with snapshot isolation, a weaker form of this guarantee

Durability

- Once a txn is committed, any data it has written will not be forgotten even if there are hardware faults/db crashes
    - typically means the db has been written to non-volatile storage, and that a WAL is used

Weak Isolation Levels

- guarantees on: what can a read-only transaction see in the presence of concurrent-writes?

Read committed

- no dirty reads: when reading from the db, you only see data that has been committed
    - this is useful because

        * in case a txn has to update many objects, and has made a few dirty writes without commiting the txn, another txn may see some of the writes and not others
        * if a txn aborts and the its writes need to be rolled back, then a txn may see data that later needs to be rolled back
- no dirty writes: when writing to the db, you only overwrite data that has been committed
    - typically by delaying second write until the first write's txn has committed or aborted
- popular isolation lvel -- default in PostgresSQL
- Dirty writes typically implemented using a row-level/per-object lock. Only one txn can hold a row lock at a time
- the same approach could also work for dirty reads, but acquiring read locks does not work well in practice
    - because one long running txn can force mnay other txns to wait, even if the other txns just want to read a value (not write anything t the db)
    - this harms response times
- so in practice, most dbs just rememebr both committed and dirty value for a row, and return the committed value to queries
- Problem 1: read skew/non-repeatable reads: a timing anomaly where reading state across multiple objects -- temporary inconsistency
    - t1 read A ----- t2 modify A ----- t2 modify B --- t2 commit ----- t1 read B --- t1 commit
        * the sum across two separately updated objects - when t1 reads A and B in its txn - may not tie out to expected

Snapshot Isolation/Repeatable Read

- temporary inconsistency may be ok for many use cases but for some it is not
    - backups
    - analytic queries and integrity checks
    - in these cases, it is non-sensical to observe parts of the db at different points in time
- This is where snapshot isolation helps: the idea is that each txn reads from a *consistent snapshot* in the db -- the txn sees all the data committed in the db at the start of the txn, even if the data is subsequently changed by another txn
- good for long running queries
- *readers never block writers and writers never block readers*
- various in-progress transactions typically need to observe the state of the db at differnet points in time -- so the db needs to maintain various versions of the objects side-by-side -- known as MVCC or multi-version concurrency control
- but this cant prevent
    - lost updates

  - write skew

Problem 1: Lost Updates

- guarantees on: what can write transactions do in the presence of concurrent-writes?
- The lost update problem: common in read-modify-write cycles -- can lose a modification when two txns do this concurrently, since the txns wont account for each other
  - incrementing a counter
  - making local changes to a complex value liek adding an element to a list in a JSON document
  - two users updating a wiki page

Ways around this:
1. Atomic write operations: Atomic writes are usually implemented by taking an exclusive lock on the object when it is read, so no other interleaving txn can read during the txn

Problem 2: Write-skew is a generalization of the lost-update problem

- It can happen when two transactions concurrently read the same objects, and then update some of these objects (different transactions update different objects)
  - vs dirty writes or lost updates, where different transactions update the same object
  - typically less obvious that a conflict occurred here
- neither a dirty write not a lost update since the updates are happening to different objects!
- typical flow is: read a value - make a decision based on this value - write something
- but by the time the write happens, the condition is no longer true
- eg: meeting room booking system

Serializability

The strongest isolation level -- it guarantees that even though txns may execute in parallel, the end result is as if they had executed one a time, serially, without any concurrency

There is usually a performance cost to doing this.

3 approaches to this:
1. Actual Serial Execution
2. 2-phase locking
3. Optimistic concurrecny control

Actual Serial Execution

The best way to handle concurrency is to not have concurrent txns at all. Execute txns one oafter the other, in serial order, on a single thread!

This is a somewhat recent development

- RAM became cheap enough to store entire active datasets in memory - so txns can execute much faster
- Db designers realized that online systems txns are usually short and only make a small number of reads and writes
  - by contrast, long-runing analytic queries are typically read-only and can be run on a consistent snapshot outside the serial execution loop

However, if you were to use a single thread single txn excution mechanism, you cannot be waiting on I/O -- this would result in very poor performance

- so isntead of waiting for network communcation between the app and the db server to specify queries/logic, these systems typicall run *stored procedures* that are pre stored/written in the db logic
  - this brings its own complications -- typically harder to manage/debug db code
- partitioning brings some challenges too -- can each CPU get its own piece of the dataset

Within some constraints, serial execution is viable

- txns are small and fast -- it only takes one slow txn to stall all txn processing
- limited to use cases where the active dataset can fit in memory

Two phase locking

Several txns are able to read the same object as long as no one is writing to it. But as soon as someone is writing to an object, exclusive access is required.

Writers block other readers and writers, while readers block writers. This is unlike SI, where *writers dont block readers and readers dont block writers*

If txn A has read an object and txn B wants to write to it, B must wait till A commits or aborts --
so B cant change the object behind A's back

If A has written an object and B wants to read it, B must wait till A commits or aborts -- there is
no concept of reading an older version of the object

If a txn wants to read an aobject - it must first acquire the lock in shared mode

- can share with other txns that need a shared lock, but if another txn has an exclusive lock, they must all wait

if a txn wants to write an object - if must first acquire the lock in exclusive mode

- no other txn can hold the lock at the same time

Two phases to the locking protocol: in the first phase, as the txn executes, it acquires locks. Then at the end of the txn, it releases locks

Reduces concurrency and therefore performance

- also some overhead to acquiring all locks
- one txn can acquire a lot of lock

Serializable Snapshot Isolation