Transactions:
A transaction is a way for an application to group several reads and writes together into a logical unit. Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (commit) or it fails (abort, rollback).

Issues with databases:
• The database software or hardware may fail at any time (including in the middle of a write operation).
• The application may crash at any time (including halfway through a series of operations).
• Interruptions in the network can unexpectedly cut off the application from the database, or one database node from another.
• Several clients may write to the database at the same time, overwriting each other's changes.
• A client may read data that doesn't make sense because it has only partially been updated.
• Race conditions between clients can cause surprising bugs.

** Database provide safety garuntees i.e. ACID**
** All relational databases today, and some nonrelational databases, support transactions. **

ACID:
Atomicity: (Abortability)
If the writes are grouped together into an atomic transaction, and the transaction cannot be completed (committed) due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction.

Atomicity simplifies this problem: if a transaction was aborted, the application can be sure that it didn't change anything, so it can safely be retried.

** atomicity is not about concurrency **

consistency:
The idea of ACID consistency is that you have certain statements about your data (invariants) that must always be true.
If a transaction starts with a database that is valid according to these invariants, and any writes during the transaction preserve the validity, then you can be sure that the invariants are always satisfied.

** Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application.**

Isolation:
Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other.
isolation as serializability, which means that each transaction can pretend that it is the only transaction running on the entire database. The database ensures that when the transactions have committed, the result is the same as if they had run serially (one after another), even though in reality they may have run concurrently.

To achieve isolation in transactions, we have certain levels from strictness to flexible, called Isolation levels..

Durability:
Durability is the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.
Peception of the same may vary with single and distributed database.
perfect durability does not exist

Extra Notes about transactions

* Atomicity can be implemented using a log for crash recovery and isolation can be implemented using a lock on each object (allowing only one thread to access an object at any one time). *

* increment operation, which removes the need for a read-modify-write cycle*

* compare-and-set operation, which allows a write to happen only if the value has not been concurrently changed by someone else *

* the indexes also need to be updated every time you change a value *

* can be aborted and safely retried if an error occurred. *

Should we retry transaction after failure:

- retrying the transaction causes it to be performed twice if network failure occurs after successful transaction.
- If the error is due to overload, retrying the transaction will make the problem worse, not better. To avoid such feedback cycles, you can limit the number of retries.
- If the transaction also has side effects outside of the database, those side effects
- may happen even if the transaction is aborted. (Email will go twice if its in transaction and transaction got retried)

Transaction Types:

Single Object: one write, but may have huge data (json update)

Multi Object: group of writes, dependent or independent

* Many distributed datastores have abandoned multi-object transactions because they are difficult to implement across partitions *

Transaction Isolations:

- *Strong Isolation levels*
  - *Serializable Isolation (run transactions serially)*
- *Week Isolation levels (nonSerializable)*
  - ***read committed***
    - *When reading from the database, you will only see data that has been committed*
      - *Solving dirty read problem i.e. :*
        - *Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a dirty read*
        - *Solution are:*
          1. *row level locking*
          2. *Hold both (old and new value), give old one untill transaction is commited.*
    - *When writing to the database, you will only overwrite data that has been committed*
      - *Solving dirty writes problem i.e. :*
        - *what happens if the earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value? This is called a dirty write. If transactions update multiple objects, dirty writes can lead to a bad outcome.*
        - *Solution is row level locking*
    - It is default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL
    - Does not prevent the race condition between two counter increments
    - Issues not solved in read commited
      - nonrepeatable read (bad read only for one time)

  - **Snapshot isolation**
    - The idea is that each transaction reads from a consistent snapshot of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.
      - Solving bad read on long running queries and backups, i.e.
        - During the time that the backup process is running, writes will continue to be made to the database. Thus, you could end up with some parts of the backup containing an older version of the data, and other parts containing a newer version.
        - Solution is maintains several versions of an object side by side, this technique is known as multi-version concurrency control (MVCC)

      - Solving nonrepeatable read
        - (Rules for MVCC) an object is visible if both of the following conditions are true:
          - At the time when the reader's transaction started, the transaction that created the object had already committed.
          - The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.
    - supported by PostgreSQL, MySQL with InnoDB storage engine, Oracle, SQL Server
    - readers never block writers, and writers never block readers
    - In Oracle it is called serializable, and in PostgreSQL and MySQL it is called repeatable read

** storage engines that support snapshot isolation typically use MVCC for their read committed isolation level as well. **
** A typical approach is that read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction. **
** in MVCC When a transaction is started, it is given a unique, always-increasing vii transaction ID (txid). Whenever a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer**
How indexes are handled in MVCC ??

----------- Some more problems and their solutions --------------------

lost update problem:

- *The lost update problem can occur if an application reads some value from the data-base, modifies it, and writes back the modified value (a read-modify-write cycle). If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification.*
- *Solutions:*
    - *Atomic write operations- concurrency safe - UPDATE counters SET value = value + 1 WHERE key = 'foo'; - cursor stability*
    - *Explicit locking ==> what if requirement involves some logic that you cannot sensibly implement as a data base query. Do this then*
    - *Compare-and-set ==> PDATE wiki_pages SET content = 'new content' WHERE id = 1234 AND content = 'old content';*