Atomicity

Characteristics Atomicity Main article: Atomicity (database systems) Atomicity requires that each transaction is " all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible (" atomic"), and an aborted transaction does not happen. Consistency Main article: Consistency (database systems)

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors do not violate any defined rules. Isolation Main article: Isolation (database systems)

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i. e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction. [citation needed] Durability Main article: Durability (database systems) Durability means that once a transaction has been committed, it will remain o, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute,

the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory. Examples The following examples further illustrate the ACID properties. In these examples, the database table has two fields, A and B, in two records. An integrity constraint requires that the value in A and the value in B must sum to 100.

The following SQL code creates a table as described above: CREATE TABLE acidtest (A INTEGER, B INTEGER CHECK (A + B = 100)); Atomicity failure Assume that a transaction attempts to subtract 10 from A and add 10 to B. This is a valid transaction, since the data continue to satisfy the constraint after it has executed. However, assume that after removing 10 from A, the transaction is unable to modify B. If the database retained A's new value, atomicity and the constraint would both be violated. Atomicity requires that both parts of this transaction, or neither, be complete.

Consistency failure Consistency is a very general term which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency. Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins.

If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and

the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed. Isolation failure To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data.

One of the two must wait until the other completes in order to maintain isolation. Consider two transactions. T1 transfers 10 from A to B. T2 transfers 10 from B to A. Combined, there are four actions: T1 subtracts 10 from A T1 adds 10 to B. T2 subtracts 10 from B T2 adds 10 to A. If these operations are performed in order, isolation is maintained, although T2 must wait. Consider what happens if T1 fails half-way through. The database eliminates T1's effects, and T2 sees only valid data. By interleaving the transactions, the actual order of actions might be: A - 10, B - 10, B + 10, A + 10.

Again, consider what happens if T1 fails. T1 still subtracts 10 from A. Now, T2 adds 10 to A restoring it to its initial value. Now T1 fails. What should A's value be? T2 has already changed it. Also, T1 never changed B. T2 subtracts 10 from it. If T2 is allowed to complete, B's value will be 10 too low, and A's value will be unchanged, leaving an invalid database. This is known as a write-write failure, because two transactions attempted to write to the same data field. Durability failure Assume that a transaction transfers 10 from A to B. It removes 10 from A. It then adds 10 to B.

At this point, a " success" message is sent to the user. However, the changes are still queued in the disk buffer waiting to be committed to the disk. Power fails and the changes are lost. The user assumes (understandably) that the

changes have been made. Implementation Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time. There are two popular families of techniques: write ahead logging and shadow paging.

In both cases, locks must be acquired on all information that is updated, and depending on the level of isolation, possibly on all data that is read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database. [dubious – discuss] That allows the database to return to a consistent state in the event of a crash. In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits. Locking vs multiversioning Many databases rely upon locking to provide ACID capabilities.

Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that are read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes.

Two phase locking is often applied to guarantee full isolation. [citation needed] An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified

version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks. I. e. , writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction.

User A gets a consistent view of the database even if other users are changing data. One implementation relaxes the isolation property, namely snapshot isolation. Distributed transactions Guaranteeing ACID properties in a distributed transaction across a distributed database where no single node is responsible for all data affecting a transaction presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes, because of a failure on another node.

The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not. [citation needed] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transactio