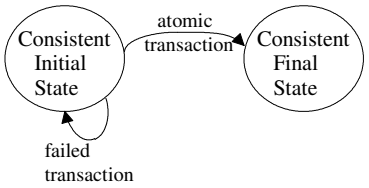
1. Failure Atomicity: Atomic Transaction has two "legal" outcomes: all or nothing



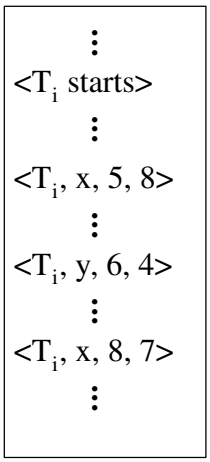
Failed or aborted transaction - must leave state in initial start

Undo logging - Every time a new entity value is written, an entry is appended to the undo log showing how its previous value can be restored.

Transaction Durability - (committed) data entity values and undo log must be stored in persistent storage (nonvolatile - disk or flash memory)

Types of Log Entries	**Description**
<T_i starts>	Indicates the start of transaction T_i
<T_i, entity, old value, new value>	Indicates update of entity by transaction T_i
<T_i commits>	Indicates that T_i has committed
<T_i aborts>	Indicates that T_i has aborted or failed
<checkpoint>	Indicates point where all log entries, then all updated entity values were flushed from main memory to persistent storage

Write-ahead log



a) When a transaction aborts, how can the log be used to undo its updates?

b) **Before** a transaction updates an entity's value in persistent storage, the *write-ahead log* entry must be written to persistent storage. Why is this order required?

c) **Before** the commit entry is written to persistent storage, all entities must be written back into persistent storage. Why is this order required?

d) After a crash during the *recovery process*, how are transactions that were in the middle of execution (and need aborting) identified?

Since persistent (disk) storage is so slow, entity values are read into memory when first accessed with changes to entities and the undo log updated buffered in memory. Periodically, the (1) undo log updated are flushed to the write-ahead log, (3) followed by changes to **all the entities**, and finally (3) writing <checkpoint> being written to the write-ahead log.

e) On recovery from a crash what needs to be done with a transaction that <commits> before the last <checkpoint>?

f) On recovery from a crash what needs to be done with a transaction that <commits> after the last <checkpoint>?

g) On recovery from a crash what needs to be done with a transaction that does not <commit> after the last <checkpoint>?

2. Two-phase locking ensures serializability at the expense of concurrency and throughput since transactions may be forced to wait for locks.

a) Why is two-phase locking not a problem if all transactions are brief with few entities (e.g., airline reservation system)?

b) Why is two-phase locking not a problem if all transactions are long-running and read-only even if they involve almost all the entities in the database? For example, transactions that data-mining historical business data for useful patterns (e.g., airline analyzing last year's passenger travel history to determine possible new routes).

c) Why would a mixture of the above transaction types likely perform badly if ran concurrently?

d) Many large businesses solve this problem of lots of little updates with some big data analysis by maintaining two separate database systems:

- operational system - mission critical little data updates performed on the fly
- data warehouse - slightly out-of-date copy (e.g., updated nightly) of the operational system used for long-running or interactive data analysis

Why is a slightly out-of-date copy of the database typically okay for data warehouse applications?

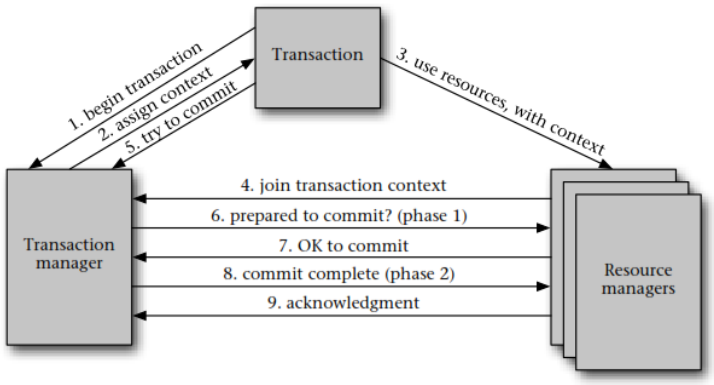
3. Some databases increase transaction concurrency (i.e., *reduce isolation*) by relaxing of serializability (two-phase locking) to improve performance. Microsoft SQL Server and IBM DB2 default to *read committed* mode where exclusive-locks are unchanged, but transactions acquire a shared lock before each read and release it immediately after the read. (a) How does this increase transaction concurrency?

b) Read committed allows for a strange phenomena called *nonrepeatable read* where a transaction reads an entity and later reads the same entity again, but gets a different value. How is *nonrepeatable read* possible in read committed mode?

4. Oracle and PostgreSQL relax serializability using a *multiversion concurrency control* (*MVCC*) approach called *snapshot isolation*. Each write stores the new value for an entity in a difference location than the old value. Thus, a read action need not read the most recent value, but it can read an older version. In snapshot isolation, a transaction can read all entities (other than those it has written) from the version that was most recently committed when the transaction started. Any writes done after the transaction started whether committed or not are ignored.
a) Explain why snapshot isolation provides repeatable reads.

b) Get an example to illustrate that snapshot isolation is not truly serializable?

5. *Two-phase commit* coordinates multiple subsystems working together on share transactions.



1. When a new transaction begins, it registers with the transaction manager.
2. In return, the transaction manager assigns an identifying *transaction context*.
3. Whenever the transaction uses the services of a resource manager, it presents its transaction context. (If the resource manager subcontracts to another resource manger, it passes the transaction context along.)
4. When a resource manager sees a new transaction context for the first time, it registers with the transaction manager as being involved in that transaction. This is known as joining the transaction.
5. When the transaction wishes to commit, it contacts the transaction manager.

6. The transaction manager knows all the involved resource managers because of their earlier join messages. The transaction manager starts phase one by asking each of those resource managers whether it is prepared to commit.
7. When a resource manager is asked to prepare to commit, it checks whether it has any reason not to. (For example, a database system might check whether any consistency constraints were violated.) If the resource manager detects a problem, it replies to the transaction manager that the transaction should be aborted. If there is no problem, the resource manager first makes sure the transaction's updates are all stored in persistent storage (for example, in redo log records). Then, once this is complete, the resource manager indicates to the transaction manager that the transaction can commit, so far as this resource manager is concerned.
8. The transaction manager waits until it has received replies from all the resource managers. If the replies indicate unanimous agreement to commit, the transaction manager logs a commitment record and notifies all the resource managers, which starts phase two.
9. When a resource manager hears that the transaction is in phase two of commitment, it logs its own commit record and drops any exclusive locks it has been holding for the transaction. Once the transaction is in phase two, there is no possibility it will abort and need to perform undo actions. Even if the system crashes and restarts, the transaction manager will see its own commitment log record and go forward with phase two.

   Each resource manager then sends an acknowledgment to the transaction manager, indicating completion of the phase two activity. When all of these acknowledgments are received, the transaction manager logs completion of the commit. That way, after a crash and restart, it will know not to bother redoing phase two.