**Lecture 8**

**Transaction Processing**

Atomicity is handled by the commit and abort mechanisms

* **commit** ends tx and ensures all changes are saved
* **abort** ends tx and *undoes* changes "already made"

Durability is handled by implementing stable storage, via

* redundancy, to deal with hardware failures
* logging/checkpoint mechanisms, to recover state

Isolation is handled by concurrency control mechanisms

* two possibilities: lock-based, timestamp-based
* various levels of isolation are possible (e.g. serializable)

### Transaction Terminology

To describe transaction effects, we consider:

* **READ** - transfer data from "disk" to memory
* **WRITE** - transfer data from memory to "disk"
* **ABORT** - terminate transaction, unsuccessfully
* **COMMIT** - terminate transaction, successfully

Relationship between the above operations and SQL:

* **SELECT** produces **READ** operations on the database
* **UPDATE** and **DELETE** produce **READ** then **WRITE** operations
* **INSERT** produces **WRITE** operations

|  |  |  |
| --- | --- | --- |
| *RT(X)* |  | read item *X* in transaction *T* |
| *WT(X)* |  | write item *X* in transaction *T* |
| *AT* |  | abort transaction *T* |
| *CT* |  | commit transaction *T* |

### Schedules

A schedule gives the sequence of operations from *≥ 1* tx

Serial schedule for a set of tx's *T1* .. *Tn*

* all operations of *Ti* complete before *Ti+1* begins

E.g.   *RT1(A)   WT1(A)   RT2(B)   RT2(A)   WT3(C)   WT3(B)*

Concurrent schedule for a set of tx's *T1* .. *Tn*

* operations from individual *Ti*'s are interleaved

E.g.   *RT1(A)   RT2(B)   WT1(A)   WT3(C)   WT3(B)   RT2(A)*

* dirty read:   
  reading data item currently in use by another tx
* nonrepeateable read:   
  re-reading data item, since changed by another tx
* phantom read:   
  re-reading result set, since changed by another tx

### Serializability

Two formulations of serializability:

* conflict serializibility
  + i.e. conflicting R/W operations occur in the "right order"
  + check via precedence graph; look for absence of cycles
* view serializibility
  + i.e. read operations *see* the correct version of data
  + checked via VS conditions on likely equivalent schedules

View serializability is strictly weaker than conflict serializability.

A screenshot of a cell phone

Description automatically generated

For transaction isolation, PostgreSQL

* provides syntax for all four levels
* treats read uncommitted as read committed
* repeatable read behaves *like* serializable
* default level is read committed

### Concurrency Control

Approaches to concurrency control:

* Lock-based
  + Synchronise tx execution via locks on relevant part of DB.
* Version-based   (multi-version concurrency control)
  + Allow multiple consistent versions of the data to exist.   
    Each tx has access only to version existing at start of tx.
* Validation-based   (optimistic concurrency control)
  + Execute all tx's; check for validity problems on commit.
* Timestamp-based
  + Organise tx execution via timestamps assigned to actions.

### Lock-based Concurrency Control

Lock table entries contain:

* object being locked   (DB, table, tuple, field)
* type of lock: read/shared, write/exclusive
* FIFO queue of tx's requesting this lock
* count of tx's currently holding lock   (max 1 for write locks)

Lock and unlock operations *must* be atomic.

Lock upgrade:

* if a tx holds a read lock, and it is the only tx holding that lock
* then the lock can be converted into a write lock

Synchronise access to shared data items via following rules:

* before reading *X*, get read (shared) lock on *X*
* before writing *X*, get write (exclusive) lock on *X*
* a tx attempting to get a read lock on *X* is blocked if another tx already has write lock on *X*
* a tx attempting to get an write lock on *X* is blocked if another tx has any kind of lock on *X*

These rules alone do not guarantee serializability.

### Two-Phase Locking

To guarantee serializability, we require an additional constraint:

* in every tx, all *lock* requests precede all *unlock* requests

Each transaction is then structured as:

* growing phase where locks are acquired
* action phase where "real work" is done
* shrinking phase where locks are released

Clearly, this reduces potential concurrency ...