IT 775
Database Technology

SQL-DML

Locking

Locking

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking

DBS manages concurrency using locks on DB resources

- e.g., tables, rows, columns …

granularity of locks is a tradeoff

- many small specific locks maximizes concurrency
- fewer coarse locks minimizes transaction mgmt. overhead
- table or page level common compromises

Locking - Basic Rules

If transaction has shared lock on item, can read but not update item.

If transaction has exclusive lock on item, can both read and update item.

Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.

Exclusive lock gives transaction exclusive access to that item.

Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.

Locking

locking is often implicit or a combo of implicit & explicit

- implicit
  - many DBMS implicitly acquire locks within transactions
  - resources accessed by statement locked for stmt duration
  - multi-statement trans. implicitly lock resources 'til commit/abort
  - implicit locking by 2 step commit trans. solves some problems
    - dirty reads, lost updates, inconsistent summaries
- explicit
  - some trans. mgrs support explicit locking for multiple stmts
  - can help with deadlock, serializability, higher isolation levels
  - also usable outside of transactions

Locking –– Enforce Concurrency Rules

DB concurrency control forces app to lock DB items before accessing them

Locking systems come in many forms

binary lock: holder has exclusive access

read/write lock:

- read lock holder shares access with other readers, not writers
- write lock holder has exclusive access
  - except for don't care readers
- read lock holder who wants to update must secure write lock

must ReadLock or WriteLock item to read it

must WriteLock item to (over)write it

must Unlock item when finished with it

ReadLock holder can request upgrade to WriteLock for update

WriteLock holder can downgrade to ReadLock

Two Phase Locking Protocol

Phase 1 transaction acquires all necessary locks -- growth phase

Phase 2 it releases lock as it finishes with item -- shrinking phase

guarantees serializable schedules

if *all* active applications follow the protocol

can reduce concurrency

PostgreSQL locking

most locks implicit, based on isolation level

- PostgreSQL uses MVCC (multiversion concurrency control) instead of explicit locking or most purposes

explicit locks provided for more detailed control

- e.g., deadlock prevention

Preventing Lost Update Problem using 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing Uncommitted Dependency Problem using 2PL

Time	T_3	T_4	$\mathbf{bal_x}$
t_1		begin_transaction	100
t_2		write_lock($\mathbf{bal_x}$)	100
t_3		read($\mathbf{bal_x}$)	100
t_4	begin_transaction	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	100
t_5	write_lock($\mathbf{bal_x}$)	write($\mathbf{bal_x}$)	200
t_6	WAIT	rollback/unlock($\mathbf{bal_x}$)	100
t_7	read($\mathbf{bal_x}$)		100
t_8	$\mathbf{bal_x} = \mathbf{bal_x} - 10$		100
t_9	write($\mathbf{bal_x}$)		90
t_{10}	commit/unlock($\mathbf{bal_x}$)		90

Preventing Inconsistent Analysis Problem using 2PL

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Deadlock Prevention

pessimistic --

all applications get all locks up front

if it can’t, it releases all it did get and retries later

some applications may wait a long time

apps request needed locks in prescribed system-wide order

all applications lock X & Y in same order,

X & Y can't cause deadlock

optimistic --

let app lock items as needed

detect deadlock when it occurs & recover

- abort an uncommitted transaction to let rest proceed, restart it later

Locking Resources in Order

both schedules follow 2 phase locking
however one can deadlock & one can't

A_1	A_2	
read–lock(Y)	read–lock(X)	can deadlock
write–lock(X)	write–lock(Y)	
…	…	
read–lock(Y)	write–lock(Y)	deadlock free
write–lock(X)	read–lock(X)	but no concur.
…	…	