Database Systems

Concurrency

H. Turgut Uyar Şule Öğüdücü

2002-2012

License

Topics

Transactions

- a group of operations may have to be carried out together
  - finishing some operations while failing on others might cause inconsistency
- no guarantee that multiple operations will all be finished
  - at least return to the state before the changes

Definition
transaction: a logical unit of work

Transaction Example

Example (transferring money from one bank account to another)

```sql
UPDATE ACCOUNTS SET BALANCE = BALANCE - 100
  WHERE ACCOUNTID = 123

UPDATE ACCOUNTS SET BALANCE = BALANCE + 100
  WHERE ACCOUNTID = 456
```


Transaction Properties

- atomicity
  - all or nothing
- consistency
  - moving from one consistent state to another
- isolation
  - whether operations of an unfinished transaction affect other transactions or not
- durability
  - when a transaction is finished, its changes are permanent even if there is a system failure

Transaction Start/End

starting a transaction

```
BEGIN [ WORK | TRANSACTION ]
```

finishing a transaction

```
COMMIT [ WORK | TRANSACTION ]
```

cancelling a transaction

```
ROLLBACK [ WORK | TRANSACTION ]
```


Transaction Example

Example

```
  BEGIN TRANSACTION
  ON ERROR GOTO UNDO
  UPDATE ACCOUNTS SET BALANCE = BALANCE - 100
    WHERE (ACCOUNTID = 123)
  UPDATE ACCOUNTS SET BALANCE = BALANCE + 100
    WHERE (ACCOUNTID = 456)
  COMMIT
  ...

UNDO:
  ROLLBACK
```


Recovery

- consider a system failure during a transaction
  - buffer cache has not been flushed to the disk
- how to guarantee durability?
  - it should be possible to derive the data from other sources in the system
  - internal level

Transaction Log

- the log keeps the values of every affected tuple before and after every operation
- write-ahead log rule: the log must be flushed to the physical medium before the transaction is committed
- accessing records in the log is sequential by nature

Checkpoints

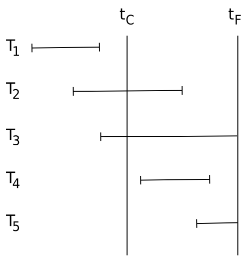
- create checkpoints in the log at certain intervals
- flush buffer cache to the physical medium
- note the checkpoint in the log
- note the continuing transactions

Recovery Lists

- after the failure, which transactions will be undone, which transactions will be made permanent?
  - create two lists: *undo* (U), *redo* (R)
- t_C: last checkpoint in the log
  - add the transactions which are active at t_C to U
- scan records from t_C to end of log
  - add any starting transaction to U
  - move any finishing transaction to R

Recovery Example

Example



- t_C:
  $U = \{T_2, T_3\}$ $R = \emptyset$
- T_4 started:
  $U = \{T_2, T_3, T_4\}$ $R = \emptyset$
- T_2 finished:
  $U = \{T_3, T_4\}$ $R = \{T_2\}$
- T_5 started:
  $U = \{T_3, T_4, T_5\}$ $R = \{T_2\}$
- T_4 finished:
  $U = \{T_3, T_5\}$ $R = \{T_2, T_4\}$

Recovery Process

- scan records backwards from end of log
  - undo the changes made by the transactions in U
- scan records forwards
  - redo the changes made by the transactions in R

Two-Phase Commit

- different source managers
  - different undo / redo mechanisms
- modifications on data that reside on different source managers
  - either commit in all source managers
    or rollback in all source managers
- coordinator

Protocol

- coordinator tells all participants to flush
  all data regarding the transaction to the physical medium
- coordinator tells all participants to start the transaction
  and report back the result
  - if all participants report success, coordinator decides to commit
  - if one or more participants report failure,
    coordinator decides to rollback
- coordinator informs the participants about the decision

References

Required Reading: Date

- Chapter 15: Recovery

Concurrency

- problems that might arise due to simultaneous transactions:
- lost update
- uncommitted dependency
- inconsistent analysis

Lost Update

Example

Transaction A	Transaction B
…	…
RETRIEVE p	…
…	…
…	RETRIEVE p
…	…
UPDATE p	…
…	…
…	UPDATE p
…	…

Uncommitted Dependency

Example

Transaction A	Transaction B
…	…
…	UPDATE p
…	…
RETRIEVE p	…
…	…
…	ROLLBACK
…	

Inconsistent Analysis

Example (sum of accounts: acc1=40, acc2=50, acc3=30)

Transaction A	Transaction B
…	…
RETRIEVE acc1 (40)	…
RETRIEVE acc2 (90)	…
…	…
…	UPDATE acc3 (30 → 20)
…	UPDATE acc1 (40 → 50)
…	COMMIT
…	…
RETRIEVE acc3 (110)	
…	

Conflicts

- A reads, B reads
  - no problem
- A reads, B writes
  - non-repeatable read (inconsistent analysis)
- A writes, B reads
  - dirty read (uncommitted dependency)
- A writes, B writes
  - dirty write (lost update)

Locking

- transactions lock the tuples they work on
  - shared lock (S)
  - exclusive lock (X)
- they release the locks when they are done

Lock Requests

lock type compatibility matrix

	X	S	-
X	N	N	Y
S	N	Y	Y

- if exclusive lock, all lock requests are denied
- if shared lock:
  - exclusive lock requests are denied
  - shared lock requests are granted

Locking Protocol

- the transaction requests a lock depending on the operation it wants to perform
  - promote a shared lock to an exclusive lock
- if the request cannot be granted, it starts waiting
  - it continues when the transaction that holds the lock releases it
  - starvation

Lost Update

Example

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
...	RETRIEVE p (S+)
...	...
UPDATE p (X-)	...
wait	...
wait	UPDATE p (X-)
wait	wait

Uncommitted Dependency

Example

Transaction A	Transaction B
...	...
...	UPDATE p (X+)
...	...
RETRIEVE p (S-)	...
wait	...
wait	ROLLBACK
RETRIEVE p (S+)	
...	

Inconsistent Analysis

Example (sum of accounts: acc1=40, acc2=50, acc3=30)

Transaction A	Transaction B
...	...
RETRIEVE acc1 (S+)	...
RETRIEVE acc2 (S+)	...
...	...
...	UPDATE acc3 (X+)
...	UPDATE acc1 (X-)
...	wait
RETRIEVE acc3 (S-)	wait
wait	wait

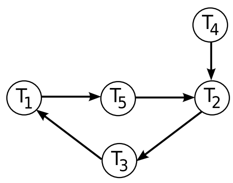
Deadlock

Definition
deadlock:
transactions waiting for each other to release the locks

- almost always between two transactions
- countermeasures:
  - detecting and solving
  - preventing

Solving Deadlocks

Example



- wait graph
- choose a victim and kill it

Preventing Deadlocks

- every transaction has a starting timestamp
- if the lock request of transaction A
  conflicts with a lock held by transaction B:
  - wait-die: A waits if it is older than B,
    otherwise it dies
    A is rolled back and restarted
  - wound-wait: A waits if it is younger than B,
    otherwise it wounds B
    B is rolled back and restarted
- the timestamp of a restarted transaction is not changed

Lock Statements

shared lock

```
SELECT query FOR SHARE
```

exclusive lock

```
SELECT query FOR UPDATE
```


Isolation Levels

- if isolation is decreased, concurrency can be increased
- various isolation levels:

- serializable
- repeatable read
- read committed
- read uncommitted

Serializability

- *serial execution*:
  a transaction starts only after another is finished
- serializable: the result of concurrent execution is always
  the same as one of the serial executions

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$

- first A, then B: $x = 22$
- first B, then A: $x = 21$

Two-Phase Locking

- two-phase locking:
  after any lock is released
  there will be no more new lock requests
  - expansion phase: gather locks
  - contraction phase: release locks
- two-phase strict locking:
  all locks are released at the end of the transaction
- *If all transactions obey the two-phase locking protocol,*
  all concurrent executions are serializable.

Read Committed

- only exclusive locks are held until end of the transaction

Example

Transaction A	Transaction B
...	...
RETRIEVE p (S+)	...
...	...
release lock	...
...	...
...	UPDATE p (X+)
...	COMMIT
RETRIEVE p (S+)	

Phantoms

Definition
phantom: when the query is executed again, new tuples appear

Example
- transaction A computes the average of a customer's account balances: $\frac{100+100+100}{3} = 100$
- transaction B creates new account with balance 200 for the same customer
- transaction A computes again: $\frac{100+100+100+200}{4} = 125$

Setting Isolation Levels

Statement
```
SET TRANSACTION ISOLATION LEVEL
  [ SERIALIZABLE | REPEATABLE READ |
    READ COMMITTED | READ UNCOMMITTED ]
```


Isolation Level Problems

isolation level	dirty read	non-repeatable read	phantom
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Locking Granularity

- locking relations instead of tuples
  - even the entire database
- if granularity is increased, concurrency is decreased
- hard to find locks on tuples
  → first, get intent locks on relation variables

Intent Locks

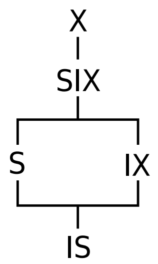
- Intent Shared (IS): the transaction intends to read some tuples
- Intent Exclusive (IX): IS + the transaction intends to write some tuples
- Shared (S): concurrent readers are allowed but no concurrent writers
- Shared + Intent Exclusive (SIX): S + IX
- Exclusive (X): no concurrency allowed on this relation

Lock Requests

lock compatibility matrix

	X	SIX	IX	S	IS	-
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y

Lock Precedence



- for a shared lock on a tuple, at least an IS lock on the relation
- for an exclusive lock on a tuple, at least an IX lock on the relation

Locking Statements

Statement

```
LOCK [ TABLE ] table_name
     [ IN lock_mode MODE ]
```

- lock modes:
  - `ACCESS SHARE`
  - `ROW SHARE`
  - `ROW EXCLUSIVE`
  - `SHARE UPDATE EXCLUSIVE`
  - `SHARE`
  - `SHARE ROW EXCLUSIVE`
  - `EXCLUSIVE`
  - `ACCESS EXCLUSIVE`

References

Required Reading: Date

- Chapter 16: Concurrency