Database Systems

Concurrency

H. Turgut Uyar Şule Öğüdücü

2002-2010

License

Topics

Transactions

- a group of operations to be carried out together
  - doing one operation while omitting the other might cause inconsistency

Definition

transaction:

a logical unit of work

Transaction Example

Example (transferring money from one bank account to another)

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

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


Transaction Management

- no guarantee that a group of operations will be carried out together
  - we should at least be able to return to the state before the changes

Transaction Properties

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

Transaction Start/End

starting a transaction

BEGIN [**WORK** | **TRANSACTION**]

successfully ending a transaction

COMMIT [**WORK** | **TRANSACTION**]

unsuccessfully ending 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

- system failure during a transaction
  - buffer cache has not been flushed to the disk
- how to guarantee durability?

Transaction Log

- data can be derived from some other source in the system
  - internal level
- the values of every tuple before and after the operation is noted in the log
  - *write-ahead rule*:
    the log must be flushed to the physical medium before the transaction is finished

Checkpoints

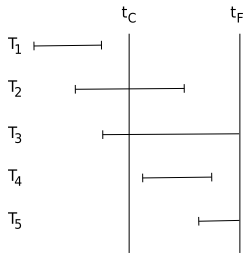
- create checkpoints in the log at certain intervals
  - flush buffer cache to the physical medium
  - note the checkpoint:
    continuing transactions

Recovery Lists

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

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 from end of log backwards
  - undo the operations of the transactions in the undo list
- scan records forwards
  - redo the operations of the transactions in the redo list

Two-Phase Commit

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

Protocol

- coordinator tells all participants to flush the 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 text: Date

- Chapter 15: Recovery

Concurrency

- 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

	-	S	X
S	Y	Y	N
X	N	N	N

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

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

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

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

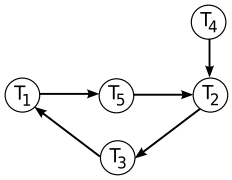
Deadlock

Definition
deadlock:
transactions are 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:
  - serializable
  - repeatable read
  - read committed
  - read uncommitted

Serializable

- *serial execution*: one transaction starts after the other is finished

Example

- $x = 10$
- transaction A: $x = x + 1$
- transaction B: $x = 2 * x$
- first A, then B: $x = 22$
- first B, then A: $x = 21$

Serializability

- serializable:
  the result of concurrent execution is always the same as one of the serial executions
- *if all transactions obey the two-phase locking protocol, all concurrent executions are serializable*

Read Committed

- only exclusive locks are held until end of transaction

Example

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

Phantoms

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

Example

- transaction A computes the average of a customer's account balances:
  $\frac{100+100+100}{3} = 100$
- transaction B creates a 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

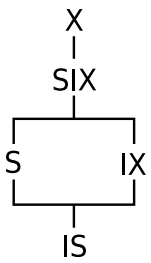
- Intent Shared (IS):
  the transaction intends to read some tuples
- Intent Exclusive (IX):
  IS + the transaction intends to write some tuples
- Shared (S):
  concurrent readers are allowed but no concurrent writers
- Shared + Intent Exclusive (SIX):
  S + IX
- Exclusive (X):
  no concurrency allowed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

Lock Requests

lock compatibility matrix

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

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 text: Date

- Chapter 16: Concurrency