Transactions 3

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Overview

- Transaction Isolation and Atomicity
- Transaction Isolation Levels
- Implementation of Isolation Levels
- Transactions as SQL Statements
- Assignments

Recoverable Schedules

- Transactions may fail in the middle of execution.
- Consider the following schedule:

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

T_7 is dependent on T_6 because read(A) saw write(A)

Partial schedule: no commit or abort for T_6

- This is an example of *nonrecoverable* schedule.
  - T_6 fails before it commits
  - Because T_6 fails, it should be aborted.
  - However, T_7 already saw T_6 committed, so T_7 cannot abort.
  - Thus, it is not possible to recover from the failure.
- **Recoverable schedule**: if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.

Cascading Rollbacks

- In some cases, a single transaction failure leads to a series of transaction rollbacks (**cascading rollback**)
- Consider the following schedule:

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

- If T_8 fails, T_9 and T_{10} must also be rolled back.
  - Not desirable, because it can take significant amount of work.

Cascadeless Schedules

- **Cascadeless schedules** — schedules where cascading rollbacks cannot occur
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Overview

- Transaction Isolation and Atomicity
- Transaction Isolation Levels
- Implementation of Isolation Levels
- Transactions as SQL Statements
- Assignments

Weak Levels of Consistency

- Serializability is useful concept to maintain consistency of database while executing transactions concurrently.
  - Ensuring serializability may allow too little concurrency
- Some applications are willing to live with weaker levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate

Isolation Levels

- The SQL standard specifies several isolation levels:

Highest

- **Serializable** — default; ensures serializable execution.
  - However, some database systems implement this isolation level in a manner that may allow nonserializable executions in some cases. (e.g., snapshot isolation by oracle or postgres prior to version 9)
- **Repeatable read** — only committed data can be read and further requires that repeated reads of same data item by a transaction must return same value.
  - However, a transaction may not be serializable with respect to other transactions
- **Read committed** — only committed data can be read, but does not require repeatable reads; successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lowest

All isolation levels above disallow dirty writes, which is writes to a data item that has already been written by another transaction that has not yet committed or aborted

Serializability vs Performance

- An application designer may decide to use weaker isolation level for better system performance.
  - Serializability may force transactions to wait for other transactions or to abort due to deadlocks (Ch18)
- In the real-world, most major database systems run at a weaker isolation level by default, which can be explicitly set at database level or at start of transaction
  - Oracles, Postgres: Read Committed
  - MySQL: Repeatable Read
- Also, some systems actually implement a weaker level of isolation for the serializable level
  - Snapshot isolation is called "serializable" mode in Oracle and PostgreSQL versions prior to 9.1, which may cause confusion with the "real serializability" mode.
- Many means to implement isolation levels
  - Helpful to understand some details of implementation to avoid or minimize the chance of inconsistency

Overview

- Transaction Isolation and Atomicity
- Transaction Isolation Levels
- Implementation of Isolation Levels
- Transactions as SQL Statements
- Assignments

Concurrency Control

- Concurrency-control policies ensure that only acceptable schedules are generated
  - An example concurrency-control policy:
    - A transaction acquires a lock on the entire database before it starts and releases the lock after it has committed.
      - No other transaction is allowed to acquire the lock and must wait
  - The example above will generate serial schedules only, because only one transaction can execute at a time
    - Are serial schedules recoverable/cascadeless? (Yes)
  - The example above provides a poor degree of concurrency
- The goal of concurrency-control policies is to provide a high degree of concurrency while only generate schedules that are:
  - either conflict or view serializable
  - recoverable and preferably cascadeless

Implementation of Isolation Levels

- Concurrency-Control Mechanisms to implement isolation levels:
  - Locking
    - Lock may provide mutual exclusion on accessing database or data items
    - Lock for the entire database leads to poor degree of concurrency
    - Lock for each data item should be hold for long enough
    - Two-phase locking protocol can ensure serializability (Ch18)
  - Timestamps
    - Assigns timestamp for each transaction, typically when a transaction begins
    - Data items store two timestamps
      - Read timestamp
      - Write timestamp
    - Timestamps are used to detect out of order accesses
  - Multiple versions of each data item
    - Allow transactions to read from a “snapshot” of the database (**snapshot isolation** for an example)

Implementation of Isolation Levels

- Snapshot isolation overview:
  - Each transaction is given its own version (or snapshot) of the database when it begins
    - Read sees the data in the snapshot
    - Update is made to the snapshot – later, applied to “real” database
  - Transactions updating some data item can commit if there is no other transaction that has modified the same data items.
    - Otherwise, transactions cannot commit but abort.
- In snapshot isolation, read never need to wait unlike locking and read-only transaction cannot be aborted
  - Significant performance improvement compared to locking if most transactions are read-only
- Limitation is too much isolation
  - Concurrent updates in transactions may not be seen by each other
    - Can result in inconsistent database state due to nonserializable execution (Ch18)

Overview

- Transaction Isolation and Atomicity
- Transaction Isolation Levels
- Implementation of Isolation Levels
- Transactions as SQL Statements
- Assignments

Transactions as SQL Statements

- Our simple model so far only considers read and write.
  - Here write allows to update the value of data items assumed to exist
  - Yet, in the real world, data items may be created (i.e. inserted) and deleted
    - Writes can be either insert, update, or delete
  - Also, reading some data items are not trivial by looking at SQL statements.
    - Consider the following SQL query (Transaction T1)

select *ID, name*
from *instructor*
where *salary* > 90000

Which data items get accessed is depending on the Where clause

Choi	70000
Einstein	100000
Wu	90000
Brandt	95000

Phantom Phenomenon

Choi	70000
Einstein	100000
Wu	90000
Brandt	95000

- Consider concurrent execution of the following transactions
  - Transaction T1:

    select *ID, name*
    from *instructor*
    where *salary* > 90000
  - Transaction T2:

    insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000);
- Do they conflict?
  - Consider T1 → T2
    - It seems not, because T1 accesses only tuples containing 'Choi', 'Einstein', 'Wu', and 'Brandt'
  - Now, consider T2 → T1
    - There exists conflict, because T1 accesses the tuple containing 'James' as well now.
  - So, there exists actual conflict which may or may not be revealed depending on the timing.
    - This is called **phantom phenomenon**.

James	100000

"Phantom" data

Transactions as SQL Statements

- Consider the following example:
  - T1 uses the index to find data items that satisfy the condition in the where clause
  - Before T1 finishes its search, T2 inserts a new data item
  - Before T2 adds the corresponding index entry for the new data item, T1 finishes the search
    - Although T2 has already written the “phantom” data before T1 finishes reading, T1 may not see it because the index entry is not updated on time.
- Only considering each data item is not enough
  - Information used to find tuples must be considered for concurrency control as well
    - Index-locking protocols maximizing concurrency, ensuring serializability in spite of inserts, deletes and predicates in queries are discussed in Ch18.4.3

Predicate Locking

Choi	70000
Einstein	100000
Wu	~~90000~~ 81000
Brandt	95000

- Consider followings:
  - Transaction T1:

    select *ID, name*
    from *instructor*
    $\boxed{\textbf{where } \textit{salary} > 90000}$
  - Transaction T3 (Wu's salary = 90000):

    update *instructor*
    set *salary* = *salary* * 0.9
    where *name* = 'Wu'
- If the conflict is determined only based on low-level data item accesses, conflict may or may not exist for the example above:
  - If query processing of T1 searches through each tuple in the table, there exists conflict between T1 and T3
  - If index is used to find data items satisfying the condition in T1's where clause, there is no conflict
- Alternative concurrency control approach may look at predicate: "**predicate locking**" (Ch18.4.3)

Overview

- Transaction Isolation and Atomicity
- Transaction Isolation Levels
- Implementation of Isolation Levels
- Transactions as SQL Statements
- Assignments

Assignments

- Reading: Ch 17.7, 17.8, 17.9, Note 17.2, 17.10
- Practice Excercises: 17.7, 17.8, 17.9, 17.10, 17.11

Solutions to the Practice Excercises:
https://www.db-book.com/Practice-Exercises/index-solu.html

For 17.9, solution has an error. T_2 withdraws $200 from the saving account

The End