CS315: Principles of Database Systems
Database Transactions

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2013-14

Tue, Fri 1530-1700 at CS101

ACID properties

- A transaction is a logical unit of a program
- To preserve data integrity, a database must follow four properties
  1. Atomicity: Either all operations of a transaction are reflected or none are reflected
  2. Consistency: If a database is consistent before the execution of the transaction, it must be consistent after it
  3. Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of others, i.e., to a transaction, it must seem that either any other transaction has completed execution or has not started execution at all
  4. Durability: After a transaction finishes successfully, the changes must be permanent in the database despite subsequent failures
- Together, these four properties are called the ACID properties

Transaction model

- A transaction is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity

Transaction model

- A transaction is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: read and write
- read(x): reads a database item named x
- write(x): writes to a database item named x
- Database program uses the same name x for the variable

Transaction model

- A transaction is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: read and write
- read(x): reads a database item named *x*
- write(x): writes to a database item named *x*
- Database program uses the same name *x* for the variable
- A transaction can read or write a data item only once
- Conflicts with read and write

Transaction model

- A transaction is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: read and write
- read(x): reads a database item named *x*
- write(x): writes to a database item named *x*
- Database program uses the same name *x* for the variable
- A transaction can read or write a data item only once
- Conflicts with read and write
    - RAW: read-after-write
    - WAR: write-after-read
    - WAW: write-after-write

Example

- Transfer 50 rupees from account *a* to account *b*
- read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)
- Atomicity:

Example

- Transfer 50 rupees from account *a* to account *b*
- read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)
- Atomicity: *a* must not be debited without crediting *b*
- Consistency:

Example

- Transfer 50 rupees from account *a* to account *b*
- read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)
- Atomicity: *a* must not be debited without crediting *b*
- Consistency: Sum of *a* and *b* remains constant
- Isolation:

Example

- Transfer 50 rupees from account *a* to account *b*
- read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)
- Atomicity: *a* must not be debited without crediting *b*
- Consistency: Sum of *a* and *b* remains constant
- Isolation: If another transaction reads *a* and *b*, it must see one of the two states – before or after the transaction, otherwise $a + b$ is wrong
- Durability:

Example

- Transfer 50 rupees from account *a* to account *b*
- read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)
- Atomicity: *a* must not be debited without crediting *b*
- Consistency: Sum of *a* and *b* remains constant
- Isolation: If another transaction reads *a* and *b*, it must see one of the two states – before or after the transaction, otherwise $a + b$ is wrong
- Durability: If *b* is notified of credit, it must persist even if the database crashes

Problems in transactions

- Lost update: Update by one transaction is overwritten by another transaction

Problems in transactions

- Lost update: Update by one transaction is overwritten by another transaction
- Temporary update or Dirty read: A transaction updates and then fails; another transaction reads it before it is reverted to original value

Problems in transactions

- Lost update: Update by one transaction is overwritten by another transaction
- Temporary update or Dirty read: A transaction updates and then fails; another transaction reads it before it is reverted to original value
- Incorrect summary: One transaction is updating values while other is computing an aggregate on them

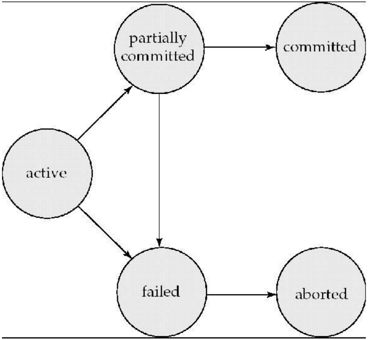
Causes of transaction failures

Causes of transaction failures

- System crash
  - Memory is lost
- System error
  - Divide by zero
- Exceptions
  - Insufficient account balance
- Concurrency enforcement
  - Deadlock detection
- Disk crash
  - Persistency fails
- Physical problems
  - Power failure, fire

Transaction state

- Active: transaction is executing
- Partially committed: after last statement has been executed
- Failed: when execution cannot proceed
- Committed: after successful completion
- Aborted: transaction has been rolled back and it has been ensured that there is no effect of the transaction



Shadow database scheme

- Recovery management system of a database ensures that atomicity and durability properties are maintained
- Shadow database scheme enforces atomicity
  - A *shadow copy* of the database is made before any transaction
  - All updates are made on the shadow copy
  - If the transaction finishes successfully, the database pointer is updated to the shadow copy
  - If the transaction fails, the old database pointer which points to the original database is maintained
- Inefficient for large databases
- Cannot efficiently handle concurrent transactions

Log

- Log or journal keeps track of all transaction operations
- Log enforces atomicity and durability
  - Log is maintained on disk
  - Log is periodically backed up to archival storage to guard against disk failures
- For a transaction *T*, following log records are maintained
  - (start, T)
  - (write, T, x, old, new)
  - (read, T, x, val)
  - (commit, T)
  - (abort, T)

Log

- Log or journal keeps track of all transaction operations
- Log enforces atomicity and durability
  - Log is maintained on disk
  - Log is periodically backed up to archival storage to guard against disk failures
- For a transaction T, following log records are maintained
  - (start, T)
  - (write, T, x, old, new)
  - (read, T, x, val)
  - (commit, T)
  - (abort, T)
- Recovery using logs
  - Undo possible by traversing log *backward* and setting database items to *old* values
  - Redo possible by traversing log *forward* and setting database items to *new* values

Commit point

- A transaction reaches its commit point when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log

Commit point

- A transaction reaches its commit point when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log
- A transaction needs to rollback if there is a (start, T) entry but no (commit, T) entry
  - It may also need to be rolled back if there is a (abort, T) entry
  - Undo operations need to be performed

Commit point

- A transaction reaches its commit point when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log
- A transaction needs to rollback if there is a (start, T) entry but no (commit, T) entry
  - It may also need to be rolled back if there is a (abort, T) entry
  - Undo operations need to be performed
- Before a transaction reaches its commit point, any portion of the log not yet written to disk must be flushed – this is called force-writing of the log
  - Ensures that redo operations can be done successfully

Concurrency

- Multiple transactions should be able to run concurrently
- Advantages
  - Increased processor and disk utilization leading to better *throughput*: one is using CPU, other is doing disk I/O
  - Reduced average response time: short transactions finish earlier and do not wait behind long ones
- Concurrency control schemes achieve isolation
- Must ensure correctness of concurrent executions
- Serializability imposes notion of correctness