CS 5330/7330

Transaction Processing Overview

Motivation

- Operations on databases (e.g. SQL commands)
  - Queries : select … from … where
  - Insertions : insert … values …
  - Deletions : delete … where …
  - Updates : update … where …
  - Create tables, change attributes etc.
- These are basic operations on tables
- But are they “too basic” in real life?

Motivation

- Consider a database for bank accounts
- Basic operations (in the eye of the customers)
  - Withdraw
  - Deposit
  - Transfer
  - Dividend
- Each basic operations contains multiple database operations

Motivation

- Example : Transfer $k from x to y (Method 1)
    1. Find tuple for x’s account (database query)
    2. Read x’s account info into main memory
    3. Check if x have at least $k
    4. Subtract $k from x’s account
    5. Write x’s new balance back to the database (database update)
    6. Find tuple for y’s account (database query)
    7. Read y’s account info into main memory
    8. Add $k to y’s account
    9. Write y’s new balance to the database (database update)

Motivation

- One need to maintain
  - Consistency/Correctness
  - Efficiency
- Correctness/consistency : The right amount of money being transferred
  - Easy to check for normal operations
  - But what if
    - System crashes
    - Multiple users want to update same data

Motivation

System crashes, case 1

1. Find tuple for x's account (database query)
2. Read x's account info into main memory
3. Check if x have at least $k
4. Subtract $k from x's account
5. Write x's new balance back to the database (database update)

System crashes! ---------------------------- *System crashes!*

6. Find tuple for y's account (database query)
7. Read y's account info into main memory
8. Add $k to y's account
9. Write y's new balance to the database (database update)

- What is the database like now?
- What happen if we don't do anything about it?

Motivation

System crashes, case 2

1. Find tuple for x’s account (database query)
2. Read x’s account info into main memory
3. Check if x have at least $k
4. Subtract $k from x’s account
5. Write x’s new balance back to the database (database update)
6. Find tuple for y’s account (database query)
7. Read y’s account info into main memory
8. Add $k to y’s account
9. Write y’s new balance to the database (database update)

System crashes! -------------------------------- System crashes!

- OK?
- But what is output is being buffered?

Motivation

- Two potential problems
  - System crashes in the middle
    - Need to make sure the system is consistent after restarting
    - Some tuples may be updated by others aren’t
    - What should one do?
  - System crashes at the “end”
    - It is unclear if all changes is saved onto the disk
    - When system crashes, all the unsaved changes is lost
    - Need to ensure that all changes are reflected

Motivation

- Another problem: multiple users
- Consider another operation, dividend:

1. Find tuple for x's account (database query)
2. Find tuple for y's account (database query)
3. Read x's account info into main memory
4. Read y's account info into main memory
5. Add 1% to x's account
6. Write x's new balance back to the database (database update)
7. Add 1% to y's account
8. Write y's new balance back to the database (database update)

Motivation

- Suppose x has \$100, y has \$200
- Consider two operations
  - x transfer \$50 to y
  - Dividend
- If transfer comes before dividend
  - X : 100 -> 50 -> 50.5
  - Y : 200 -> 250 -> 252.5
- If dividend comes before transfer
  - X : 100 -> 101 -> 51
  - Y : 200 -> 202 -> 252

Motivation

- What if we want concurrent execution?
- What does concurrent mean?
- Can we concurrently run commands without any limitations?
- What is an acceptable ***schedule***?

Motivation

1. Find tuple for x's account (database query)
2. Read x's account info into main memory
3. Check if x have at least $k
4. Subtract $k from x's account
5. Write x's new balance back to the database (database update)

1. Find tuple for x's account (database query)
2. Find tuple for y's account (database query)
3. Read x's account info into main memory
4. Read y's account info into main memory
5. Add 1% to x's account
6. Write x's new balance back to the database (database update)
7. Add 1% to y's account
8. Write y's new balance back to the database (database update)

6. Find tuple for y's account (database query)
7. Read y's account info into main memory
8. Add $k to y's account
9. Write y's new balance to the database (database update)

X : 100 -> 50 -> 50.5; Y : 200 -> 202 -> 252

Acceptable to the bank, but not the customer….

Motivation

- Thus need to define an acceptable standard of consistency, **in the face of concurrent execution with other commands**
- A plausible definition:
  - "If multiple commands execute concurrently, the results must *looks like* that the commands are executed one by one (sequentially)

Motivation

- Many of the above problems can be eliminated if we
  - Disable concurrency
  - Forcing writes to disk immediately
  - Do not write anything until the end of the command
- However this leads to inefficiency
- Thus: how to get the best of both worlds…

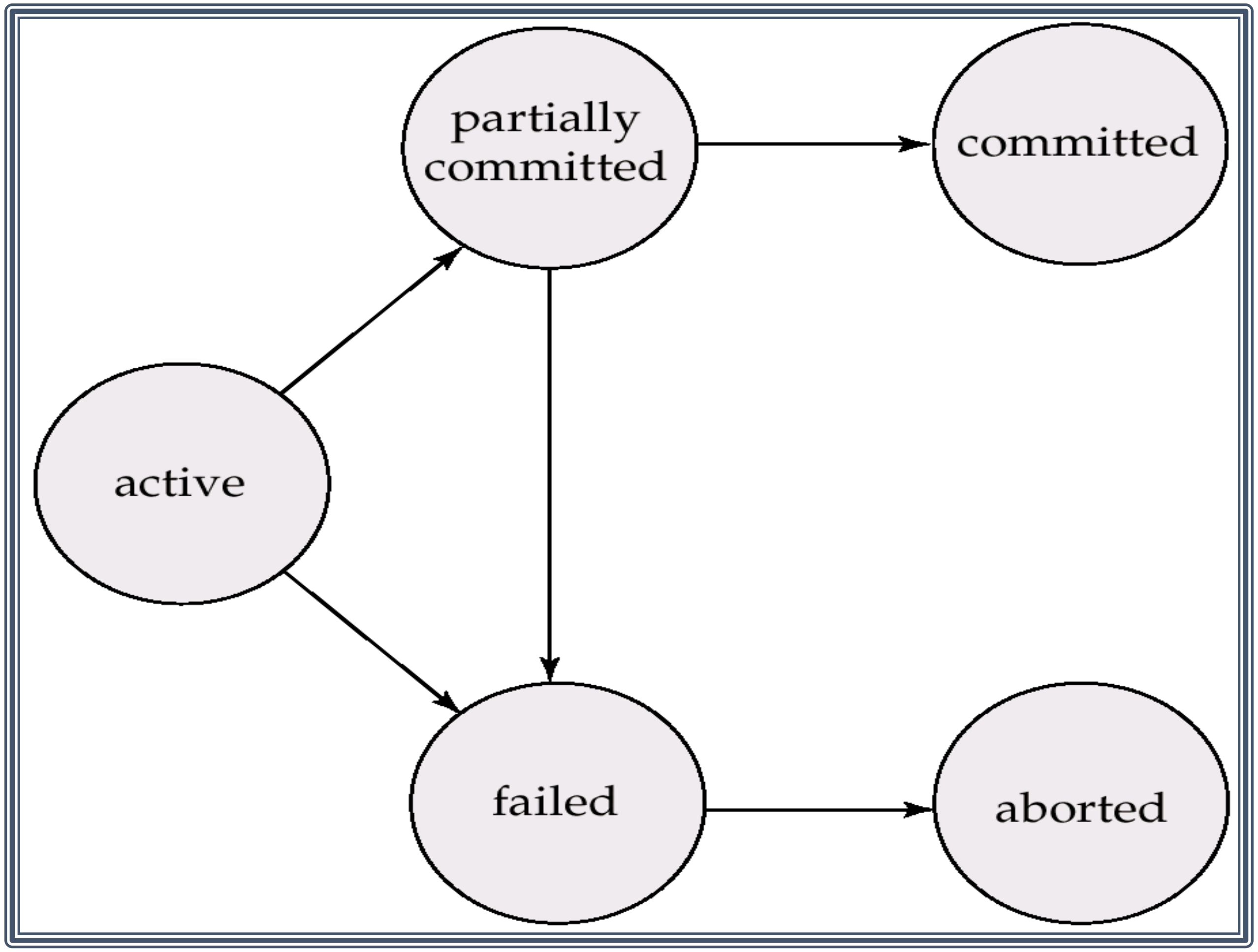
Transaction basics -- definition

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
- Can be defined as
  - A set of SQL statements
  - Stored procedures
  - Initiated by high level programming languages (Java, C++ etc.)
- Delimited by *begin transaction* & *end transaction*
- Example:
  - Begin transaction
  - X = select salary from person where name = “Chu”
  - update person set salary = x * 10 where name = “”
  - Update person set salary = x / 10 where name = “”
  - End transaction

Transaction basics -- states

- A transaction can be in any one of the 5 states:
  - **Active,** the initial state; the transaction stays in this state while it is executing
  - **Partially committed,** after the final statement has been executed.
  - **Failed,** after the discovery that normal execution can no longer proceed.
  - **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
  - **Committed,** after *successful completion*.

Transaction basics -- states



Transaction basics -- states

- A transaction need not commit immediately after its last statement
  - Why?
- It is the DBMS’s responsibility to determine which transactions can commit and which to abort
- Also, it is the DBMS’s responsibility to clean up ***(roll back)*** after a transaction aborts
- Possibility of cascade aborts

Transaction basics -- consistency

- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Transaction basics -- ACID

- Four basic properties that must be maintained
- **Atomicity** : All or nothing
- **Consistency** : Each transaction must ensure data consistency
- **Isolation** : Transactions “unaware” of other concurrent transaction
- **Durability** : Once committed, changes to database must be persistent

Transaction basics -- ACID

- **Atomicity** : All or nothing
- i.e. : Either all operations of the transaction are properly reflected in the database or none are.
- Implications
  - If the system crashes in the middle of a transaction T, when the system restarts, *before any user can use the database* again, the DBMS must ensure either
    - T is finished
    - T never started
- Which do you think is easier? Make more sense?

Transaction basics -- ACID

- **Consistency:** Each transaction must ensure data consistency
- i.e. Execution of a transaction in isolation preserves the consistency of the database.
- Thus all integrity and other constraints must be satisfied

Transaction basics -- ACID

- **Isolation** : Transactions “unaware” of other concurrent transaction
- Intermediate transaction results must be hidden from other concurrently executed transactions.
- Implications:
    - for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
    - Some level of interleaving are not allowed

Transaction basics -- ACID

- **Durability** : Once committed, changes to database must be persistent
- i.e. : After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- Implications:
  - Suppose a transaction commits, and then the system crashes. When the system restarts, *before any user can use the database* again, the DBMS must ensure that the changes made by this transaction is stored onto the disk.
- Why is this not automatically the case?

Transaction basics -- ACID

- DBMS, not the user, is required to maintain ACID properties
- The user will submit the transactions only containing the required database operations
- The DBMS will
  - Add additional operations
  - Schedule the operations
  - Introduce various data structures and algorithms

  to ensure the ACID properties hold
- If needed, the DBMS will decide when a transaction will commit and/or abort
- In many DBMS, users can specify when should a transaction commit/abort
- Roll back is also the task of the DBMS
- Need to worry about “observable external writes”

Transaction basics – DBMS support

- 2 major tasks in DBMS to handle transactions
  - ***Concurrency control***
    - Handle how concurrent transaction is executed
    - Goal: Isolation
  - ***Recovery***
    - Handle how to recover a database after a failure
    - Goal: Atomicity & Durability
  - Consistency is maintained throughout various part of the DBMS (not the focus of this course)
- Many systems rolls the two part together as a "transaction manager"

Transaction basics – DBMS support

- What is actually happening
  - Database is stored on the disk
  - DBMS allocate local memory for each transaction
  - Each transaction requests a set of tuples
  - Transaction issues read commands (e.g. select … from … where)
  - The set of tuples are read into main memory buffers
  - The value of the tuples are transferred from those buffers to the local memory for each transaction
  - Calculation and updates are done in local memory
  - The transaction issues write commands (e.g. update … set … where)
  - The values in the local memory is copied to the buffers
  - The buffers are flushed to the disk by the Operating systems (at unspecified time, unless transaction forces it)

Transaction basics – DBMS support

- What makes transaction processing tricky
  - Scheduling is hidden from the DBMS
    - DBMS cannot enforce which transaction to execute next
  - Buffer management is hidden from DBMS
    - Although the transaction write something onto the database, it is only written to the buffers, to be transferred to the disk at unspecified time
    - One can force transfer immediately, but will be very inefficient

Notation used (rest of semester)

- Database consists of objects (X, Y, Z), each of them is an integer
- Transactions are labeled T_1, T_2 etc.
- Each transaction has a set of local variables (not accessible by other transactions) in main memory. (Labelled as a1, a2, b1, b2 etc.)
- Each transaction access the database by the read() & write() command

Notation used (rest of semester)

- A read command read a database object into a local variables (a1 <- read(X))
- A write command write a local variable into the database object (write(X, a1))
- Local variables for read() & write() will not be shown if the context is clear, or if it is unimportant
- Manipulation and calculation on objects can only be done on local variables (e.g. X <- X + 1 is not allowed, but a1 <- a1 + 1 is ok)
- In some case, the local manipulation is not shown (to highlight the effects of read() and write())

Notation used (rest of semester)

- Example; (transfer, assuming overdraft is allowed)

```
1. A1 <- Read(X)
2. A1 <- A1 – k
3. Write(X, A1)
4. A2 <- Read(Y)
5. A2 <- A2 + k
6. Write(Y, A2)
```