CS162
Operating Systems and Systems Programming
Lecture 13

Transactions

October 12, 2011
Anthony D. Joseph and Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- What is a database?
- Transactions
- Conflict serializability

Note: Some slides and/or pictures in the following are adapted from lecture notes by Mike Franklin.

What is a Database

- A large **integrated collection** of data
- Models real world, e.g., enterprise
  - Entities (e.g., teams, games)
  - Relationships, e.g.,
    Cal plays against Stanford in The Big Game

Key Concept: Structured Data

- A data model is a collection of entities and their relationships
- A schema is an instance of a data model
  - E.g., describes the fields in the database; how the database is organized
- A relational data model is the most used data model
  - Relation, a table with rows and columns
  - Every relation has a schema which describes the fields in the column

Example: University Database

- Conceptual schema:
  Students(sid*: string*, name*: string*, age*: integer*, gpa*:real*)
  Courses(cid*: string*, cname*:string*, credits*:integer*)
  Enrolled(sid*:string*, cid*:string*, grade*:string*)
  FOREIGN KEY sid REFERENCES Students
  FOREIGN KEY cid REFERENCES Courses

- External Schema (View):
  Course_info(cid:string,enrollment:integer)
  Create View Course_info AS
  SELECT cid, Count (*) as enrollment
  FROM Enrolled
  GROUP BY cid

Example: An Instance of Students Relation

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

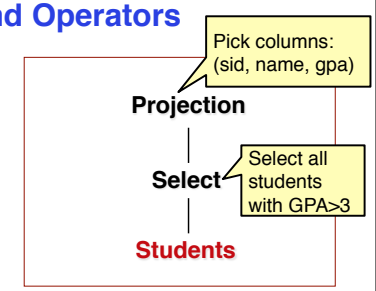
What is a Database System?

- A **Database Management System (DBMS)** is a software system designed to store, manage, and facilitate access to databases.

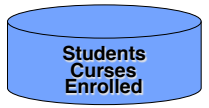
- A DBMS provides:
  - Data Definition Language (DDL)
    - » Define relations, schema
  - Data Manipulation Language (DML)
    - » Queries – to retrieve, analyze and modify data.
  - Guarantees about durability, concurrency, semantics, etc

Key Concepts: Queries, Query Plans, and Operators

```
SELECT sid, name, gpa
FROM Students S
WHERE S.gpa > 3
```



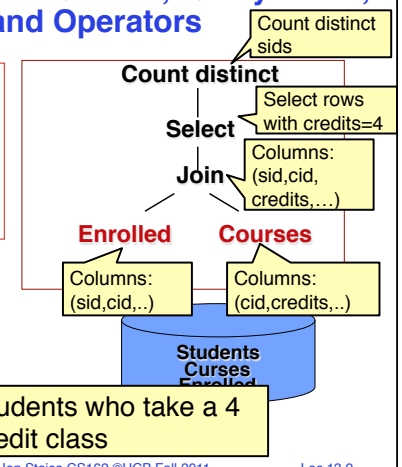
System handles query plan generation & optimization; ensures correct execution.



Select all students with GPA > 3.0

Key Concepts: Queries, Query Plans, and Operators

```
SELECT
    COUNT DISTINCT (E.sid)
FROM Enrolled E, Courses C
WHERE  E.cid = C.cid
    AND C.credits = 4
```

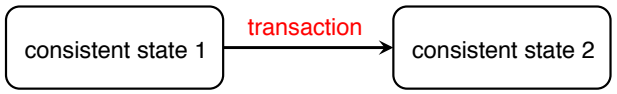


System handles query plan generation & optimization; ensures correct execution.

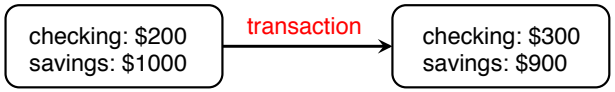
Number of students who take a 4 credit class

Key concept: Transaction

- An atomic sequence of database actions (reads/writes)
- Takes DB from one consistent state to another



Example



- Here, *consistency* is based on our knowledge of banking "semantics"
- In general, up to writer of transaction to ensure transaction preserves consistency
- DBMS provides (limited) automatic enforcement, via integrity constraints (IC)
  - e.g., balances must be >= 0

From Multiprogramming to Transactions

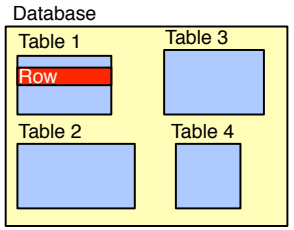
- Users would like the illusion of running their programs on the machine alone
  - Why not running the entire program in a critical section?
- Users want fast response time and operators want to increase machine utilization → increase concurrency
  - Interleave executions of multiple programs
- How can DBMS help?

Concurrent Execution & Transactions

- Concurrent execution essential for good performance
  - Disk slow, so need to keep the CPU busy by working on several user programs concurrently
- DBMS only concerned about what data is read/written from/ to the database
  - Not concerned about other operations performed by program on data
- **Transaction** - DBMS's abstract view of a user program, i.e., a sequence of reads and writes.

Locking Granularity

- What granularity to lock?
  - Database
  - Tables
  - Rows



- Fine granularity (e.g., row) → high concurrency
  - Multiple users can update the database and same table simultaneously
- Coarse granularity (e.g., database, table) → simple, but low concurrency

Transaction - Example

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance -
   100.00 WHERE name = 'Alice';

 UPDATE branches SET balance = balance -
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Alice');

 UPDATE accounts SET balance = balance +
   100.00 WHERE name = 'Bob';

 UPDATE branches SET balance = balance +
   100.00 WHERE name = (SELECT branch_name
   FROM accounts WHERE name = 'Bob');
COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** if each transaction is consistent, and the DB starts consistent, it ends up consistent
- **Isolation:** execution of one transaction is isolated from that of all others
- **Durability:** if a transaction commits, its effects persist

Atomicity

- A transaction
  - might *commit* after completing all its operations, or
  - it could *abort* (or be aborted by the DBMS) after executing some operations
- Atomic Transactions: a user can think of a transaction as always either *executing all its* operations, or *not executing any* operations at all
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions

Consistency

- Data in DBMS follows integrity constraints (ICs)
- If DBMS is consistent before transaction, it will be after
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)
  - DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements
  - Beyond this, DBMS does not understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)

Isolation

- Each transaction executes as if it was running by itself
  - Concurrency is achieved by DBMS, which interleaves operations (reads/writes of DB objects) of various transactions
- Techniques:
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability

- Data should survive in the presence of
  - System crash
  - Disk crash → need backups
- All committed updates and only those updates are reflected in the database
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

This Lecture

- Deal with **(I)solation**, by focusing on **concurrency control**
- Next lecture focus on (A)tomicity, and partially on (D)urability

Example

- Consider two transactions:
  - T1: moves $100 from account A to account B

```
T1:A := A-100; B := B+100;
```

  - T2: moves $50 from account B to account A

```
T2:A := A+50;  B := B-50;
```

- Each operation consists of (1) a read, (2) an addition/subtraction, and (3) a write
- Example: A = A-100

```
Read(A); // R(A)
A := A - 100;
Write(A); // W(A)
```


Example (cont'd)

- Database only sees reads and writes

		Database View
`T1: A:=A-100; B:=B+100;`	→	`T1:R(A),W(A),R(B),W(B)`
`T2: A:=A+50; B:=B-50;`	→	`T2:R(A),W(A),R(B),W(B)`

- Assume initially: A = $1000 and B = $500
- What is the legal outcome of running T1 and T2?
  - A = $950
  - B = $550

Example (cont'd)

```
T1: A:=A-100; B:=B+100;
T2: A:=A+50; B:=B-50;
```

Initial values:
A:=1000
B:=500

- What is the outcome of the following execution?

```
T1:R(A),W(A),R(B),W(B)
T2:                   R(A),W(A),R(B),W(B)
```

A=900 B=600 A=950 B=550

- What is the outcome of the following execution?

```
T1:                   R(A),W(A),R(B),W(B)
T2:R(A),W(A),R(B),W(B)
```

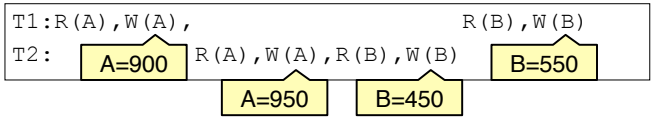
A=950 B=550 A=1050 B=450

Example (cont'd)

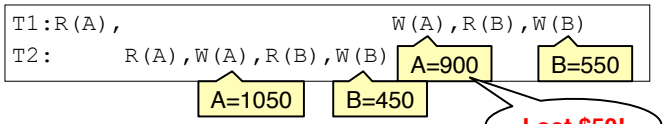
```
T1: A:=A-100; B:=B+100;
T2: A:=A+50; B:=B-50;
```

Initial values:
A:=1000
B:=500

- What is the outcome of the following execution?



- What is the outcome of the following execution?



Transaction Scheduling

- Why not run only one transaction at a time?
- Answer: low system utilization
  - Two transactions cannot run simultaneously even if they access different data
- Goal of transaction scheduling:
  - Maximize system utilization, i.e., concurency
    - » Interleave operations from different transactions
  - Preserve transaction semantics
    - » Logically the sequence of all operations in a transaction are executed atomically
    - » Intermediate state of a transaction is not visible to other tranasctions

Midterm

- Midterm: Thursday, October 13, 5-6:30pm in 155 Dwinelle
  - Up to and including lecture 11
  - Closed book, 1 cheat sheet (hand written, two sides)
- Extra office hours (right after the lecture):
  - Anthony: 5:30-6:30pm, 465 Soda Hall
  - Ion: 6:30-7:30pm, 465 Soda Hall

5min Break

Transaction Scheduling

- **Serial schedule:** A schedule that does not interleave the operations of different transactions
  - Transactions run serially (one at a time)
- **Equivalent schedules:** For any database state, the effect (on the database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule:** A schedule that is equivalent to some serial execution of the transactions
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

Anomalies with Interleaved Execution

- May violate transaction semantics, e.g., some data read by the transaction changes before committing
- Inconsistent database state, e.g., some updates are lost
- Anomalies always involves a "write"; Why?

Anomalies with Interleaved Execution

- Read-Write conflict (Unrepeatable reads)

```
T1:R(A),          R(A),W(A)
T2:     R(A),W(A)
```

- Violates transaction semantics
- Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock
  - (T1) John logs first, but waits…
  - (T2) Mary logs second and buys the TV set right away
  - (T1) John decides to buy, but it is too late…

Anomalies with Interleaved Execution

- Write-read conflict (reading uncommitted data)

```
T1:R(A),W(A),      W(A)
T2:           R(A),     …
```

- Example:
  - (T1) A user updates value of A in two steps
  - (T2) Another user reads the intermediate value of A, which can be inconsistent
  - Violates transaction semantics since T2 is not supposed to see intermediate state of T1

Anomalies with Interleaved Execution

- Write-write conflict (overwriting uncommitted data)

```
T1:W(A),          W(B)
T2:      W(A),W(B)
```

- Get T1's update of B and T2's update of A
- Violates transaction serializability
- If transactions were serial, you'd get either:
  - T1's updates of A and B
  - T2's updates of A and B

Conflict Serializable Schedules

- Two operations **conflict** if they
  - Belong to different transactions
  - Are on the same data
  - At least one of them is a write.
- Two schedules are **conflict equivalent** iff:
  - Involve same operations of same transactions
  - Every pair of **conflicting** operations is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),          R(B),W(B)
```

⇩

```
T1:R(A),W(A),     R(B),     W(B)
T2:          R(A),     W(A),     R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),     R(B),W(B)
```


Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Example:

```
T1:R(A),W(A),R(B),          W(B)
T2:               R(A),W(A),     R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),     W(B)
T2:               R(A),     W(A),R(B),W(B)
```

⇩

```
T1:R(A),W(A),R(B),W(B)
T2:                    R(A),  W(A),R(B),W(B)
```

Conflict Equivalence – Intuition (cont'd)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**
- Is this schedule serializable?

```
T1:R(A),          W(A)
T2:     R(A),W(A),
```

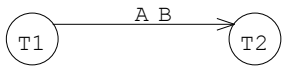

Dependency Graph

- **Dependency graph:**
  - Transactions represented as nodes
  - Edge from Ti to Tj:
    - » an operation of Ti conflicts with an operation of Tj
    - » Ti appears earlier than Tj in the schedule
- **Theorem:** Schedule is conflict serializable if and only if its dependency graph is acyclic

Example

- Conflict serializable schedule:

```
T1:R(A),W(A),          R(B),W(B)
T2:          R(A),W(A),          R(B),W(B)
```



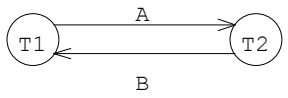
Dependency graph

- No cycle!

Example

- Conflict that is *not* serializable:

```
T1:R(A),W(A),                    R(B),W(B)
T2:          R(A),W(A),R(B),W(B)
```



Dependency graph

- Cycle: The output of T1 depends on T2, and vice-versa

Notes on Conflict Serializability

- Conflict Serializability doesn't allow all schedules that you would consider correct
  - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data
- In practice, Conflict Serializability is what gets used, because it can be done efficiently
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, …
- Two-phase locking (2PL) is how we implement it (next lecture)

Summary

- Transaction: a sequence of DB operations
- ACID:
  - Atomicity: all operations in a transaction happen, or none happens
  - Consistency: if DB starts consistent, it ends up consistent
  - Isolation: execution of one transaction is isolated from another
  - Durability: the results of a transaction persists
- Correctness criterion for transactions is "serializability".
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.