Database Systems

Transaction

Feifei Li
University of Utah
Motivated by two independent requirements

- Concurrent database access
- Resilience to system failures
Concurrent Database Access

Even more software

Select...
Update...
Create Table...
Drop Index...
Help...
Delete...

More software

DBMS

Data
Before we start: Let’s review the concept of Semaphore
The classic museum example

Requirement: at any given time, number of person inside the museum must be less than 100.

Thread 1
While (true) {
    wait for entrance signal;
    C = C+1;
}

Thread 2
While (true) {
    wait for exit signal;
    C = C-1;
}

Does this work though?
Concurrent access problem

Consider the following sequence of events:
1) T2 reads C into cache (C=99 in the eye of T2).
2) T1 reads C into cache (C=99 in the eye of T1).
3) T2 sets C=C-1 in cache (C=98 in the eye of T2).
4) T1 sets C=C+1 in cache (C=100 in the eye of T1).
5) T1 writes C back to memory (C=100 in memory).
6) T2 writes C back to memory (C=98 in memory).
The final value of C is 98!

The fundamental problem: Memory and cache access are NOT atomic!
Concurrent Access: Attribute-level Inconsistency

Update College Set enrollment = enrollment + 1000
Where cName = ‘utah’

concurrent with ...

Update College Set enrollment = enrollment + 1500
Where cName = ‘utah’

Actions involved: Get, Modify, Put. They may be interleaved!
Concurrent Access: Tuple-level Inconsistency

Update Apply Set major = ‘CS’ Where sID = 123

concurrent with ...

Update Apply Set decision = ‘Y’ Where sID = 123

Actions involved: Get, Modify, Put. They may be interleaved! Maybe only one of changes survives in the end.
Concurrent Access: Table-level Inconsistency

Update  Apply  Set  decision = ‘Y’
Where  sID  In (Select  sID  From  Student  Where  GPA > 3.9)

concurrent with ...

Update  Student  Set  GPA = (1.1) \times GPA  Where  sizeHS > 2500

Actions involved: Get, Modify, Put. They may be interleaved!
Concurrent Access: Multi-statement inconsistency

Insert Into Archive
   Select * From Apply Where decision = 'N';
Delete From Apply Where decision = 'N';

concurrent with ...

Select Count(*) From Apply;
Select Count(*) From Archive;
Concurrency Goal

Execute *sequence of SQL statements* so they appear to be running in isolation

* Simple solution: execute them in isolation
  But want to enable concurrency whenever safe to do so (i.e., interleaving actions from different threads to improve the overall performance).
Transactions in MySQL/PostgreSQL

• **Start/Begin Transaction**: starts a new transaction
• **Commit**: commits the current transaction, making its changes permanent
• **Rollback**: rolls back the current transaction, canceling its changes
• **Set autocommit = {0 off | 1 on}**: disables or enables the default autocommit mode for the current session
• **By default**: MySQL/PostgreSQL runs with autocommit mode enabled!
Transactions in MySQL

- **T1:**
  start transaction;
  insert into course values (111, `Database Systems`, `Computer Sciences`);
  commit;

- **T2:**
  start transaction;
  select * from course;
  commit
Transactions in MySQL

• Make sure to use the InnoDB storage engine to support transactions;

• To check current storage engine for a table:
  show create table Table-Name;

• To update the storage engine for a table:
  Alter table Table-Name engine=innodb;

• To set default storage engine:
  set default_storage_engine=innodb (or MyISAM)

• The default in many MySQL instances is MyISAM
Resilience to System Failures

System Crash!

Bulk Load

DBMS

Data
Resilience to System Failures

Insert Into Archive
Select * From Apply Where decision='N';
Delete From Apply Where decision='N';

Crash failure
Resilience to System Failures

Lots of updates buffered in memory

DBMS

Data

System Crash
System-Failure Goal

 Guarantee all-or-nothing execution, regardless of failures
Solution for both concurrency and failures

Transactions

A transaction is a sequence of one or more SQL operations treated as a unit

- Transactions appear to run in isolation
- If the system fails, each transaction’s changes are reflected either entirely or not at all
Solution for both concurrency and failures

A transaction is a sequence of one or more SQL operations treated as a unit. **SQL standard:**

- Transaction begins automatically on first SQL statement
- On “commit” transaction ends and new one begins
- Current transaction ends on session termination
- “Autocommit” turns each statement into transaction
Solution for both concurrency and failures

A transaction is a sequence of one or more SQL operations treated as a unit

- Transactions appear to run in isolation
- If the system fails, each transaction’s changes are reflected either entirely or not at all
(ACID Properties) **Isolation: Serializable**

**Serializability**
Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions

$\Rightarrow$ Overhead
$\Rightarrow$ Reduction in concurrency

End effect: $T_1, T_{10}, T_8, T_9, T_2$
(note that actions in different transactions may be interleaved!)
Isolation Levels

- Per transaction
- “In the eye of the beholder”
Dirty Reads

“Dirty” data item: written by an uncommitted transaction

Update College Set enrollment = enrollment + 1000
Where cName = ‘utah’

concurrent with ...

Select Avg(enrollment) From College

Dirty Reads if read this value before the 1st Transaction has committed! What happens if the 1st T rolls back after 2nd T has read this value?
Dirty Reads

“Dirty” data item: written by an uncommitted transaction

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

concurrent with ...

Select GPA From Student Where sID = 123

concurrent with ...

Update Student Set sizeHS = 2600 Where sID = 234
**Isolation Level** Repeatable Read

A transaction may not perform dirty reads
An item read multiple times cannot change value
Still does not guarantee global serializability: because of the phantom read

Transaction 1

```sql
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```sql
/* Query 2 */
INSERT INTO users VALUES (3, 'Bob', 27);
COMMIT;
```
The difference:

In Repeatable Read, there is no “Range Lock”
In Serializable “Range Lock” is Enforced!

In the SERIALIZABLE isolation mode, Query 2 would be blocked until the first transaction was committed.

In REPEATABLE READ mode, the range would not be locked, allowing the record to be inserted and the second execution of Query 1 to include the new row in its results.
Read Only transactions

- Helps system optimize performance
- Independent of isolation level

Set Transaction Read Only;
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
Isolation Levels: Summary

- **Standard default**: Serializable

- **Weaker isolation levels**
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Some systems have default Repeatable Read

- **Isolation level per transaction and “eye of the beholder”**
  - Each transaction’s reads must conform to its isolation level
Solution for both concurrency and failures

A transaction is a sequence of one or more SQL operations treated as a unit

- Transactions appear to run in isolation
- If the system fails, each transaction’s changes are reflected either entirely or not at all
ACID Properties

Atomicity
Consistency
Isolation
Durability
(ACID Properties) Isolation

Serializability
Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions

Achieved by LOCKING!
(ACID Properties) Durability

If system crashes after transaction commits, all effects of transaction remain in database.

Achieved by Logging!

System needs to REDO T2 in this case since it has “Committed.”
(ACID Properties) Atomicity

Each transaction is “all-or-nothing,” never left half done

Achieved by Logging!

System needs to UNDO T2 in this case since it has NOT “Committed” at the time of crash.
Transaction Rollback (= Abort)

- Undoes partial effects of transaction
- Can be system- or client-initiated

Each transaction is “all-or-nothing,” never left half done

Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans=‘ok’ Then Commit; Else Rollback;
(ACID Properties) Consistency

Each client, each transaction:
- Can assume all constraints hold when transaction begins
- Must guarantee all constraints hold when transaction ends

Serializability $\Rightarrow$ constraints always hold
Solution for both concurrency and failures

 Transactions

Atomicity
Consistency
Isolation
Durability