Database Transactions

- Many situations where a sequence of database operations must be treated as a single unit
  - A combination of reads and writes that must be performed “as a unit”
  - If any operation doesn’t succeed, all operations in the unit of work should be rolled back
  - Essential for correct implementation of tasks that might fail
- Also essential for DBs with concurrent access
  - An operation consisting of multiple reads may also need to see a single, consistent set of values
  - Reads should also be performed “as a unit”

Database Transactions (2)

- Databases provide <u>transactions</u> to demarcate units of work
- Issue **`BEGIN`** at start of unit of work
  - Can also say **`START TRANSACTION`**, etc.
- Issue one or more SQL DML statements within the transaction
  - DB may or may not support DDL inside transactions
- When finished, issue **`COMMIT`** command
  - Signals that the transaction is completed
- If transaction must be aborted, issue **`ROLLBACK`**
  - All changes made by the transaction are discarded

Transaction Properties

- A transaction system should satisfy specific properties
  - Called the <u>ACID</u> properties
- Atomicity
  - Either *all* the operations within the transaction are reflected properly in the database, or *none* are
- Consistency
  - When a transaction completes, the database must be in a consistent state; i.e. all constraints *must* hold
- Isolation
  - When multiple transactions execute concurrently, they must appear to execute one after the other, in isolation of each other
- Durability
  - After a transaction commits, all changes should persist, even when a system failure occurs

Bank Account Example

- Transfer $400 from account A-201 to A-305
  - Clearly requires multiple steps
- If transaction isn't atomic:
  - Perhaps only one account shows the change!
- If transaction isn't consistent:
  - Perhaps a balance goes below zero
- If transaction isn't isolated:
  - Multiple operations involving either account could result in inaccurate balances
- If transaction isn't durable:
  - If DB crashes, could end up with inaccurate balances!

Transaction Properties And You

- As a database user:
  - How atomicity, consistency, and durability are implemented is irrelevant
  - Important point is whether they're provided
    - … and how completely they are provided!
- Isolation is another matter entirely
  - Turns out to affect implementation of database applications quite extensively
  - Database users are provided many choices for how to handle transaction isolation

Transaction Isolation

- If database only has one client at a time, isolation is irrelevant
  - Client can only issue one transaction at a time
  - Two transactions can never be concurrent
- Most database applications support *many* concurrent users
  - Concurrent transactions are very frequent
- Without isolation between transactions, can end up with spurious results
- Five kinds of spurious results can occur in SQL, without proper transaction isolation

Concurrent Transaction Issues

- Dirty writes:
  - A transaction T_1 writes a value to X
  - Another transaction T_2 also writes a value to X before T_1 commits or aborts
  - If T_1 or T_2 aborts, what should be the value of X?
- Dirty reads:
  - A transaction T_1 writes a value to X
  - T_2 reads X before T_1 commits
  - If T_1 aborts, T_2 has an invalid value for X
- Nonrepeatable reads:
  - T_1 reads X
  - T_2 writes to X, or deletes X, then commits
  - If T_1 reads X again, value is now different or gone

Concurrent Transaction Issues (2)

- Phantoms
  - Transaction T_1 reads rows that satisfy a predicate P
  - Transaction T_2 then writes rows, some of which satisfy P
  - If T_1 repeats its read, it gets a different set of results
  - If T_1 writes values based on original read, new rows are left out
- Lost updates
  - Transaction T_1 reads the value of X
  - Transaction T_2 writes a new value to X
  - T_1 writes to X based on previously read value
- How can a database avoid these kinds of issues?
  - A simple answer: <u>serialize</u> all transactions
  - No two transactions can overlap, ever.
  - A *very* slow approach, but certainly works

Serialized Transactions

- Serializing all transactions is prohibitively slow
- Definite benefits for concurrent transactions:
  - Different transactions may use completely separate resources, and could run very efficiently in parallel
  - Long, slow transactions shouldn't hold up short, fast transactions
- Databases can execute transactions in a way that *appears* to be serialized
  - Transactions are sequences of read and write operations
  - Schedule these operations in a way that maintains serializability constraints

Transaction Isolation Constraints

- Serializable transaction constraint is one kind of isolation constraint
  - A very strict one, for critical operations
- Not all database applications require such strict constraints
  - Application may work fine with looser isolation constraints
  - Application might not achieve required throughput with serializable transactions
- SQL defines four transaction isolation levels for use in applications
  - Can set transactions to have a specific isolation level

Transaction Isolation Levels

- Serializable
  - Concurrent transactions produce same result as if they were run in some serial order
  - Serial order doesn't necessarily correspond to exact order that transactions were issued
- Repeatable reads
  - During a transaction, other transactions' committed changes are not visible
  - During a transaction, multiple reads of X produce same results, regardless of committed writes to X in other transactions

Transaction Isolation Levels (2)

- Read committed
  - During a transaction, other transactions' committed changes become visible immediately
  - Value of X can change during a transaction, if other transactions write to X and then commit
- Read uncommitted
  - Uncommitted changes to X in other transactions become visible immediately

Transaction Isolation Levels (3)

- Back to the undesirable transaction phenomena:
  - What does each isolation level allow?

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
serializable	NO	NO	NO
repeatable reads	NO	NO	YES
read committed	NO	YES	YES
read uncommitted	YES	YES	YES

- To specify the transaction isolation level:

```
SET TRANSACTION ISOLATION LEVEL
  { SERIALIZABLE | REPEATABLE READ |
   READ COMMITTED | READ UNCOMMITTED }
```

  - Different databases support different isolation levels!

Databases and Isolation Levels

- Most DBs implement isolation levels with locks
- At simplest level, locks are:
  - Shared, for read locks
  - Exclusive, for write locks
- Locks may have different levels of granularity
  - Row-level locks, page-level locks, table-level locks
  - Finer-grain locks allow more transaction concurrency, but demand greater system resources
  - Databases often provide multiple levels of granularity

Database Locks

- Rules for locking are carefully defined
  - What locks, or sequences of locks, satisfy the isolation constraints?
  - Can a lock be upgraded from shared to exclusive? If so, when?
- In general:
  - **`SELECT`** operations require shared locks
  - **`INSERT`**, **`UPDATE`**, **`DELETE`** operations require exclusive locks
  - In practice, gets *much* more complicated, to prevent "phantom rows" phenomenon, etc.
  - Databases vary in locking implementations and behaviors! (Read your manual…)

Locking Issues

- Some transactions are incompatible with others
  - Each transaction requires some series of locks…
  - Can easily lead to deadlock between transactions
- This can’t be avoided, because:
  - DB can’t predict what SQL commands will be issued in each transaction
  - DB can’t predict where necessary rows will appear
    - (e.g. in the same page that another transaction needs?)
- Solution:
  - Database lock managers can detect deadlocks
  - If transactions become deadlocked, one is aborted

Locking Issues (2)

- If a database application performs long or complex operations in a transaction:
  - It <u>must</u> be designed to handle situations where a transaction is aborted due to deadlock!
  - Solution is simple: just retry the operation
- Guidelines:
  - Keep transactions as short and simple as possible
  - If transactions are aborted frequently due to deadlock, the application needs to be reworked
  - Databases can usually report what commands caused the deadlock
  - Expect that deadlocks may still infrequently occur

Concurrent Reads and Writes

- Bank example:
  - Two transactions using account table
  - Repeatable-read or read-committed isolation level
  - T_1 reads balance of account A-444, gets $850
  - T_2 reads balance of account A-444, gets $850
  - T_1 adds $300 to balance of A-444
  - T_2 reads balance of A-444 again… ???
- If database stores each row in only one place then T_2 must block until T_1 commits or aborts
  - For repeatable reads, if T_1 commits then T_2 must abort!

Readers and Writers

- For certain database implementations and isolation levels, writers block readers
- Solution:
  - Keep multiple versions of each row in the database
  - If a writer updates a value:
    - A new version of the entire row is added to the database
    - Reader can continue with old version of value as long as possible
  - Writers don’t block readers anymore, in most cases
  - Writers will only block other writers to the same row

Multiversion Concurrency Control

- Called <u>multiversion concurrency control</u> (MVCC)
  - Each row has some “version” indicator
    - A timestamp or a transaction ID
  - Transactions can see a specific range of versions
    - Depends on transaction isolation level, and operations that transaction is performing
  - If a transaction reads a row that another transaction has written, reader can still proceed
    - (for read-committed and many repeatable-read scenarios)
- Yields *dramatic* performance improvements for concurrent transaction processing!
- Can make transaction isolation more confusing
  - Transactions proceed that would block without MVCC

Read-Uncommitted Example

Operation: Deposit $100 into account A-333

```
T1: BEGIN;
T1: SELECT balance FROM account                +---------+
      WHERE account_number = 'A-333';          |  850.00 |
                                               +---------+
T2: BEGIN;
T2: SELECT balance FROM account                +---------+
      WHERE account_number = 'A-333';          |  850.00 |
                                               +---------+
T1: UPDATE account SET balance = balance + 100
      WHERE account_number = 'A-333';
T2: SELECT balance FROM account                +---------+
      WHERE account_number = 'A-333';          |  950.00 |
                                               +---------+
T1: ROLLBACK;
T2: SELECT balance FROM account                +---------+
      WHERE account_number = 'A-333';          |  850.00 |
                                               +---------+
```

Read-Committed Example

Operation: Deposit $100 into account A-333

T_1: **BEGIN;**

T_1: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+---------+
|  850.00 |
+---------+
```

T_2: **BEGIN;**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+---------+
|  850.00 |
+---------+
```

T_1: **UPDATE account SET balance = balance + 100 WHERE account_number = 'A-333';**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+---------+
|  850.00 |
+---------+
```

T_1: **COMMIT;**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+---------+
|  950.00 |
+---------+
```

Repeatable-Read Example

Operation: Deposit $100 into account A-333

T_1: **BEGIN;**

T_1: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

T_2: **BEGIN;**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

T_1: **UPDATE account SET balance = balance + 100 WHERE account_number = 'A-333';**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

T_1: **COMMIT;**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

Serializable Example

Operation: Deposit $100 into account A-333

T_1: **BEGIN;**

T_1: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

T_2: **BEGIN;**

T_2: **SELECT balance FROM account WHERE account_number = 'A-333';**

```
+----------+
|  850.00  |
+----------+
```

T_1: **UPDATE account SET balance = balance + 100 WHERE account_number = 'A-333';**

- T_1 blocks on the update, because T_1 and T_2 must be completely isolated from each other!
- T_1 *must* wait for completion of T_2, since T_2 read balance of A-333 before T_1 tried to update it.

Read Issues

- Simple example:
  - Bank account A-201 jointly owned by customers A and B, with balance of $900
  - Customer A requests a loan of $800 at the bank
    - Loan amount must be less than current account balance
  - At same time, Customer B withdraws $200 from the same account
- Customer A's transaction needs the latest value
  - Value read from DB immediately goes out of date
  - Serializable transaction would prevent this, but read-committed and repeatable-read transactions allow it

Read Issues (2)

T_1: Customer A wants a loan of $800

- Customer A owns account A-201, balance $900
- Loan amount must be less than account balance

T_2: Customer B tries to withdraw $200 from A-201

- Customer B also owns account A-201

- T_1 reads account balance of $900
- T_2 subtracts $200 from account balance
- T_1 creates a new loan of amount $800
  - Bad assumption: Old value of $900 is still valid!
- Database is no longer in a consistent state
  - Bank’s business rule is violated

Read-Only Values

- Transaction T_1 needs latest value, and it must not be allowed to change until T_1 is finished!
- **`SELECT`** … **`LOCK IN SHARE MODE`** allows a transaction to mark selected values as read-only
  - Constraint is enforced until end of transaction
- Transaction T_1:

```
SELECT balance FROM account WHERE
  account_number = 'A-201' LOCK IN SHARE MODE;
```

  - T_2 cannot change balance of A-201 until T_1 is finished

Read/Write Issues

- Two banking transactions:
  - T_1 wants to withdraw all money in account A-102
  - T_2 wants to withdraw $50 from A-102
  - T_1 needs to read current balance, before it can update
  - T_2 can simply update
- T_1 reads balance of A-102
- T_2 subtracts $50 from A-102
- T_1 subtracts $400 from A-102
  - Overdraft! T_1 must roll back.

```
+----------+
|  400.00  |
+----------+
+----------+
|  350.00  |
+----------+
+----------+
|  -50.00  |
+----------+
```

- Again, prohibited by serializable transactions
  - Allowed by read-committed or repeatable-read levels

Intention to Update

- Transaction T_1 must read before its update
  - …but read lock is insufficient for purpose of T_1
  - T_1 must state intention to update row, when it reads it
  - Otherwise, T_1 will be overruled frequently
- **`SELECT`** … **`FOR UPDATE`** command allows a transaction to state intention to update

```
SELECT balance FROM account
  WHERE account_number = 'A-102'
  FOR UPDATE;
```

  - T_2 can't update A-102 until T_1 is finished

Serializable Transactions?

- Serializable transactions prevent a lot of issues
  - Serializable transactions are <u>consistent</u>
- Other isolation levels can cause some problems
  - Considered to be weak levels of consistency
- Why not serializable transactions for everything?
  - Serializable transactions are very slow for large database applications
  - Simply not scalable
  - Only certain operations run into trouble with other isolation levels
  - Can use features like **`FOR UPDATE`** as workarounds

Savepoints

- Transactions may involve a long sequence of steps
  - If one step fails, don't roll back entire transaction
  - Instead, roll back to last "good" point and try something else
- Some databases provide <u>savepoints</u>
  - Mark a savepoint in a transaction when it completes some tasks
  - Can roll back to savepoint, and continue transaction from there
- To mark a savepoint:

  `SAVEPOINT` ***`name;`***

  - Roll back to that savepoint:

    `ROLLBACK TO SAVEPOINT` ***`name;`***

  - Can release a savepoint when it becomes unnecessary:

    `RELEASE SAVEPOINT name;`

  - Commit and rollback commands work on whole-transaction level

Review

- Transaction processing is a very rich topic
  - Many powerful tools for applications to use
  - Optimizations that allow for faster throughput
- Subtle issues can arise with transactions!
  - Applications should expect that transactions might be aborted by the database
  - Sometimes operations require statements like **`SELECT`** … **`FOR UPDATE`** to work correctly
- Always read your database manual! ☺
  - What isolation levels are supported? Any variances?
  - Are **`FOR UPDATE`** / **`LOCK IN SHARE MODE`** supported?
  - Are savepoints supported?