Example

To transfer $100 from account 123 to account 456:

- check that the balance of account 123 is at least $100
- deduct $100 from account 123
- add $100 to account 456

```
IF ( SELECT balance FROM ACCOUNT WHERE acctnum=123 ) >= 100 THEN
  UPDATE ACCOUNT SET balance=balance-100 WHERE acctnum=123;
  UPDATE ACCOUNT SET balance=balance+100 WHERE acctnum=456;
END IF
```

What can go wrong?

- simultaneous transfers from account 123 can result in a negative balance
- a crash or a failure of one of the `UPDATE`s can result in a loss of money

Transactions

Things can go wrong if tasks require multiple steps, and those steps involve INSERT / DELETE / UPDATE.

- unfortunate interleaving of steps from simultaneous requests can lead to incorrect or inconsistent database state
- failure of a step (or a system crash) can lead to incorrect or inconsistent database state

```
IF ( SELECT balance FROM ACCOUNT WHERE acctnum=123 ) >= 100 THEN
  UPDATE ACCOUNT SET balance=balance-100 WHERE acctnum=123;
  UPDATE ACCOUNT SET balance=balance+100 WHERE acctnum=456;
END IF
```


Transactions

Serializability refers to the property that tasks must appear as if one is executed entirely before or after the other.

Atomicity refers to the property that either the whole task is executed, or none of it.

From the database user's perspective, these properties are achieved by designating a multi-step task to be a single *transaction*.

- SQL standard requires both serializability and atomicity, but some DBMS may allow relaxations of serializability

Transactions

```
START TRANSACTION
  statements
COMMIT
```

- COMMIT means that the transaction commits successfully
- changes become permanent

```
START TRANSACTION
  statements
ROLLBACK
```

- ROLLBACK aborts the transaction
- any changes made to the DB as part of this transaction are undone

Using Rollback

Two common scenarios –

- in a stored procedure, define an exit handler (invoked when an error occurs) which executes a rollback
- in a database application, check error codes or otherwise programmatically determine that an error occurred and issue a rollback as needed

Example

delete a sailor by name, doing nothing if there is more than one sailor with the name

```
CREATE PROCEDURE DeleteSailorByName2
  ( IN delname VARCHAR(45) )
BEGIN
  DECLARE delsid INT;
  DECLARE EXIT HANDLER FOR SQLSTATE '42000'
    BEGIN SELECT 'too many sailors!'; END;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN ROLLBACK; END;

  START TRANSACTION;

  SELECT Sid FROM SAILOR
    WHERE Sname=delname
    INTO delsid;

  DELETE FROM RESERVATION WHERE Sid=delsid;
  DELETE FROM SAILOR WHERE Sid=delsid;

  COMMIT;
END
```


Transactions

Serializability and atomicity has a cost.

- overhead of the locking mechanism used to prevent simultaneous access
- loss of concurrency because tasks are executed one at a time

Not enforcing serializability and atomicity also has a cost.

- incorrect and inconsistent database state due to *dirty reads* of data modified by a not-yet-committed operation
- inconsistent results due to *non-repeatable reads* of data modified by a committed operation

But such reads are not always harmful, and preventing problems does not always require full mutual exclusion.

Tools

- *isolation levels* allow relaxation of serializability for performance gains
  - SQL defines four isolation levels
- *locking reads* allow targeted mutual exclusion at lower isolation levels

Isolation Levels

- `SERIALIZABLE`
  - `SELECT` locks so rows accessed so no other transaction can modify them until this transaction completes
  - blocks if another transaction has uncommitted changes
  - dirty reads are not permitted
  - reads are repeatable
  - necessary when data integrity is essential (such as in banking transactions) but
    - there is a significant performance cost
    - applications must handle retrying transactions that fail

Isolation Levels

- `READ UNCOMMITTED`
  - transaction sees what has happened in other transactions, even if not committed
  - repeated `SELECT`s may return different data
  - dirty reads are permitted
  - reads are non-repeatable
  - only appropriate in limited circumstances
    - e.g. only working with data that won't be modified
    - e.g. computing summary statistics where complete accuracy isn't required and the volume of processing is so high that the performance difference is significant
  - not all DBMSes support

Isolation Levels

- `READ COMMITTED`
  - (only) committed updates are visible in this transaction
  - repeated `SELECT`s may return different data
  - dirty reads are not permitted
  - reads are non-repeatable
  - appropriate when up-to-date data is needed with each `SELECT`
    - e.g. periodic reports or inventory lists
  - the default in some DBMSes

Isolation Levels

- `REPEATABLE READ`
  - all reads within a transaction are consistent – reads are repeatable
  - first `SELECT` establishes a snapshot used by all subsequent `SELECT`s [MySQL implementation]
    - if another transaction inserts, deletes, or updates a row after that point, it won't be seen (even if committed)
  - `DELETE` or `UPDATE` may affect rows committed by another transaction, and those changes become visible
  - dirty reads are not permitted
  - applications may need to handle retrying failed transactions (not in MySQL)
  - appropriate when a complex calculation needs consistent results from multiple `SELECT`s
  - default in MySQL

Isolation Levels

```
SET [ GLOBAL | SESSION ]
    TRANSACTION ISOLATION LEVEL level
```

- `GLOBAL` sets for all subsequent sessions (but not the current one)
  - requires admin privileges to use
- `SESSION` sets for all subsequent transactions in the current session
- nothing sets for just the next transaction

See the current session isolation level with

```
SELECT @@transaction_ISOLATION
```


Locking Reads

```
IF ( SELECT balance FROM ACCOUNT WHERE acctnum=123 ) >= 100 THEN
  UPDATE ACCOUNT SET balance=balance-100 WHERE acctnum=123;
  UPDATE ACCOUNT SET balance=balance+100 WHERE acctnum=456;
END IF
```

- a negative balance can result if another transaction modifies account 123's balance between the `SELECT` and `UPDATE`
  - `SERIALIZABLE` prevents this problem, by locking the rows the `SELECT` accesses
  - `REPEATABLE READ` does not
- `SELECT … FOR SHARE` sets a shared lock on the rows read
  - other transactions can read those rows but not modify them
- `SELECT … FOR UPDATE` locks the rows read
  - blocks locking reads and modifications by other transactions

Deadlock

Deadlock occurs when two or more transactions are blocked waiting to acquire a lock held by one of the others.

- DBMS detects deadlock and terminates and rolls back at least one of the transactions involved
  - application may need to retry failed transactions

To reduce the likelihood of deadlock –

- minimize the use of locks – use higher isolation levels and locking reads only when required
- keep transactions small and short in duration – less likely to have two simultaneous ones
- take locks in a consistent order each time

MySQL Transactions

MySQL defaults –

- each statement is a transaction
- isolation level is `REPEATABLE READ`

Limitations –

- there are statements that can't be rolled back, and others that involve an implicit commit
  - e.g. data definition statements (`CREATE TABLE`, `DROP TABLE`, …)
  - check the MySQL manual for details
- triggers cannot contain `START TRANSACTION`, `COMMIT`, or `ROLLBACK`
  - check the MySQL manual for details on what happens if a trigger or the operation itself fails

Takeaways

- transactions should be considered when operations involve more than one SQL statement
  - locking reads provide targeted protection at lower isolation levels
- which isolation level to use?
  - short answer – `READ COMMITTED` and `REPEATABLE READ` (MySQL default) are most common
  - longer answer

> `InnoDB` supports each of the transaction isolation levels described here using different locking strategies. You can enforce a high degree of consistency with the default `REPEATABLE READ` level, for operations on crucial data where ACID compliance is important. Or you can relax the consistency rules with `READ COMMITTED` or even `READ UNCOMMITTED`, in situations such as bulk reporting where precise consistency and repeatable results are less important than minimizing the amount of overhead for locking. `SERIALIZABLE` enforces even stricter rules than `REPEATABLE READ`, and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

  - but there are many subtleties and implementations vary between DBMSes – for serious use, consult the MySQL documentation

https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html