Concurrency

- Enterprise-scale DBMSs are designed to host multiple databases and handle multiple concurrent connections.
- *Transactions* are designed to enable
  - Data consistency
  - Isolation of connections
Transactions

• Transactions provide "all-or-nothing" behavior
  – Either everything completes entirely or nothing is applied

• Transactions provide isolation from other transactions
  – Gives a consistent view of the data
ACID

• Atomicity
  – Transactions are atomic

• Consistency
  – Data is always in a consistent state

• Isolation
  – Multiple concurrent transactions are isolated from each other

• Durability
  – Guarantees that committed transactions are persistent
Atomicity

• Many situations require that transactions act atomically
  – Either everything done in a transaction happens or nothing does

• Example: transferring funds

  UPDATE account SET balance = balance + 50
  WHERE account_id = 123;

  UPDATE account SET balance = balance - 50
  WHERE account_id = 456;

  – Either both statements should complete or neither should
Consistency

• Transactions can never leave the database in an inconsistent state
  – We can specify constraints that the data must satisfy
    • e.g., "NOT NULL", "year >= 1789"
  – The DBMS guarantees that these constraints are not violated
Isolation

• Concurrent transactions must not impact each other
  – But sometimes transactions *conflict*
    • e.g., two simultaneous transfers on the same financial account
  – The SQL standard allows for different degrees of transaction isolation
Durability

• Guarantees that committed transactions are persistent
  – Can be difficult to implement on modern systems
    • file system caching
  – Requires writing to non-volatile storage before reporting transaction complete
Non-ACID

• Full ACID compliance can lead to performance bottlenecks
• Not every workload necessarily needs full ACID compliance
  – e.g., web search, comments page
• One of the more popular DBMSs, MySQL, is not fully ACID compliant
  – But it doesn't necessarily need to be
BEGIN – COMMIT - ROLLBACK

• Open a transaction with BEGIN
• Close a transaction with
  – COMMIT to persist all of the changes
  – ROLLBACK to abandon all of the changes
• An error in a transaction requires a rollback
BEGIN; UPDATE account SET balance = balance + 50 WHERE account_id = 123; UPDATE account SET balance = balance - 50 WHERE account_id = 456; COMMIT;
Transaction Isolation Levels

• The SQL standard defines four *transaction isolation levels*…
  – Read uncommitted (least strict)
  – Read committed
  – Repeatable read
  – Serializable (most strict)
• …based on three different *read phenomena*
  – Dirty read
  – Nonrepeatable read
  – Phantom read
Read Phenomena

• Dirty read
  – A transaction reads data written by a concurrent uncommitted transaction

• Nonrepeatable read
  – A transaction re-reads data it has previously read and finds that data has been modified (by a committed transaction)

• Phantom read
  – In a transaction, two identical queries are executed and the collection of rows returned by the second query is different from the first
Read Phenomena

• Dirty read
  – A transaction reads data written by a concurrent uncommitted transaction

BEGIN;
SELECT balance
FROM account WHERE id = 1;
(Gets 20)
...

UPDATE account SET balance = 10
WHERE id = 1;

SELECT balance
FROM account WHERE id = 1;
(Gets 10)
COMMIT;

BEGIN;

...
Read Phenomena

• Nonrepeatable read
  – A transaction re-reads data it has previously read and finds that data has been modified (by a committed transaction)

BEGIN;
SELECT balance FROM account WHERE id = 1;
(Gets 20)
...

SELECT balance FROM account WHERE id = 1;
(Gets 20)
...

BEGIN;
UPDATE account SET balance = 10 WHERE id = 1;
...
COMMIT;

COMMIT;
Read Phenomena

- Phantom read
  - In a transaction, two identical queries are executed and the collection of rows returned by the second query is different from the first

```
BEGIN;
SELECT count(*) FROM account WHERE balance > 10;  ...  (Gets 5)
...
BEGIN;
INSERT INTO account (id, balance) VALUES (4321, 20);

SELECT count(*) FROM account WHERE balance > 10;  ...  (Gets 5)
...
SELECT count(*) FROM account WHERE balance > 10;  ...  (Gets 6)
COMMIT;
```
Transaction Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Read committed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- Overhead of implementation increases significantly with increased isolation
  - Therefore, should use minimum isolation required for situation
- PostgreSQL only actually implements the last three
  - If read uncommitted is requested then read committed is actually supplied
  - Default is read committed
Isolation Errors

• A repeatable read or serializable transaction might get a concurrent access error

ERROR: could not serialize access due to concurrent update

ERROR: could not serialize access due to read/write dependencies among transactions

– Applications should be prepared to handle these errors and retry the transaction
Deadlock

- Transaction isolation can result in *deadlock*

```
BEGIN;
UPDATE account SET balance = 10 WHERE id = 1;

...

UPDATE account SET balance = 20 WHERE id = 2;
(Blocks waiting for second transaction to complete)

BEGIN;
... 
UPDATE account SET balance = 10 WHERE id = 2;
UPDATE account SET balance = 20 WHERE id = 1;
(Blocks waiting for first transaction to complete)
```
Deadlock Detection

• PostgreSQL automatically detects deadlock situations and will abort one of the blocked transactions
  – Applications should be prepared to handle these error

• However, deadlock detection is not perfect
Deadlock Avoidance

• Theoretically, deadlock can be avoided by having each transaction acquire locks in the same order
  – Unfortunately, this is not always feasible in practice