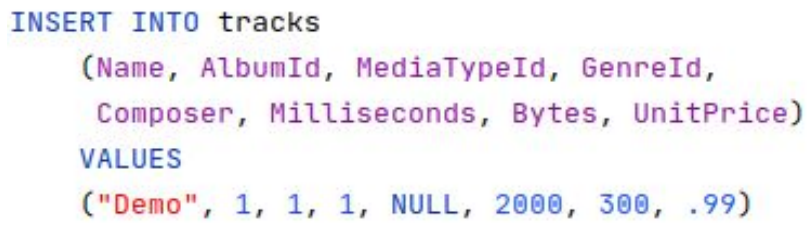
MONTANA
STATE UNIVERSITY

Transactions

Ensuring Data Consistency

Transactions In SQL

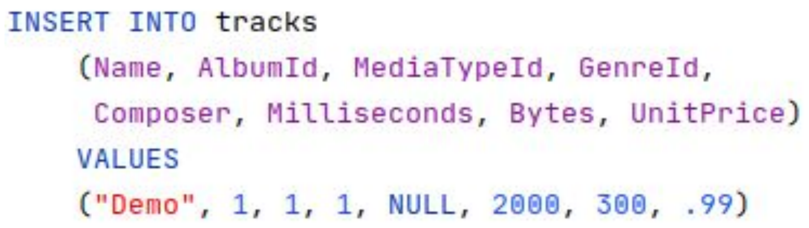
- Thus far we have treated operations as if each was independent
  - The operations are executed one at a time
  - Each one is “atomic” in that it succeeds or fails entirely



```sql
INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId,
     Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    ("Demo", 1, 1, 1, NULL, 2000, 300, .99)
```

ACID Properties

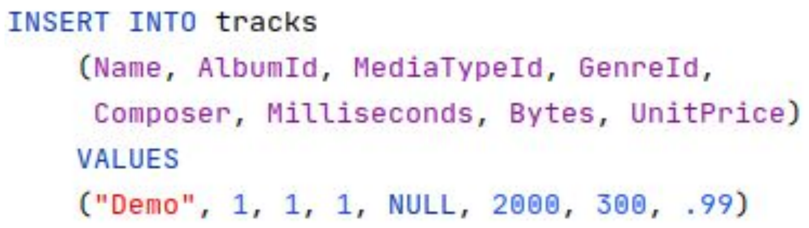
- ACID properties of databases
  - A = Atomicity
  - C = Consistency
  - I = Isolation
  - D = Durability
- *Transactions* help in ensuring these properties



```
INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId,
     Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    ("Demo", 1, 1, 1, NULL, 2000, 300, .99)
```

Atomicity

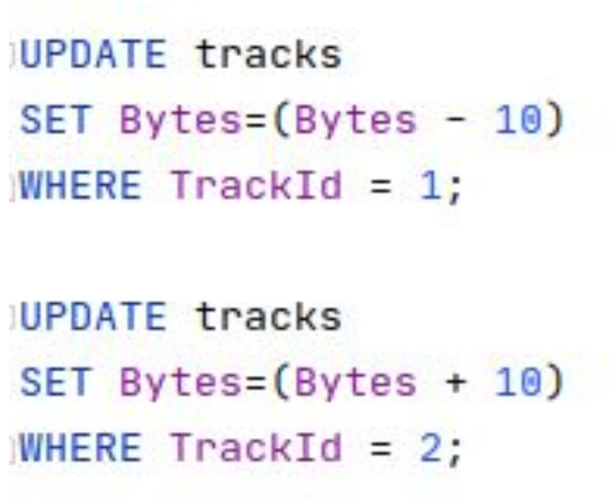
- Atomicity: an operation either completes entirely or fails entirely
  - If the DBMS goes down half way through the Track Insert, when the DBMS resumes it will be as if the insert never happened



```
INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId,
     Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    ("Demo", 1, 1, 1, NULL, 2000, 300, .99)
```

Atomicity

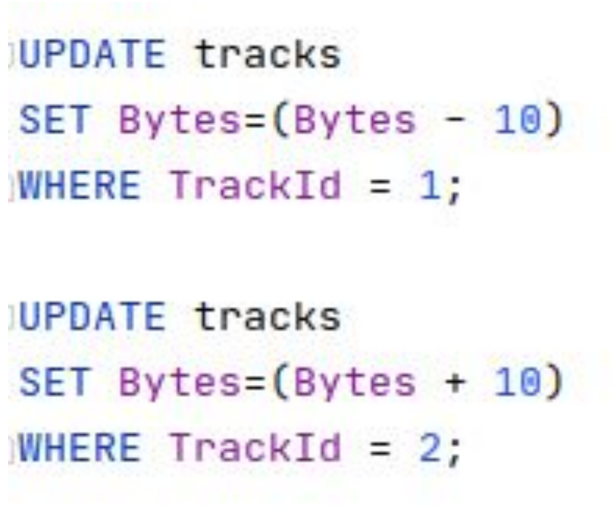
- Atomicity also can apply to multiple statements
- Here we are transferring 10 bytes from one track to another
  - Just pretend, OK!
- What happens if the DBMS goes down after the first operation but before the second?



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Atomicity

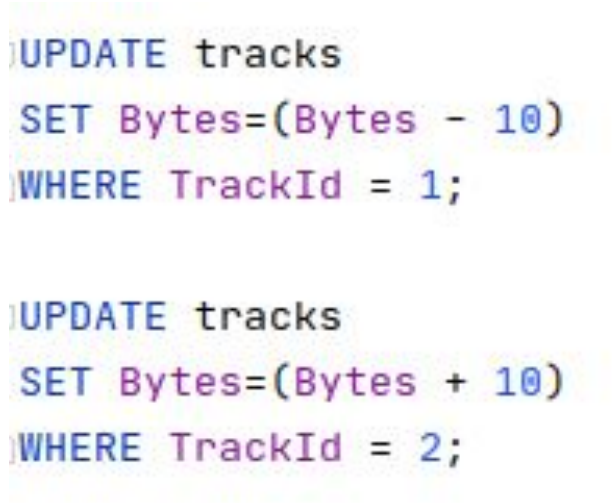
- What if this happened with bank accounts?



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Consistency

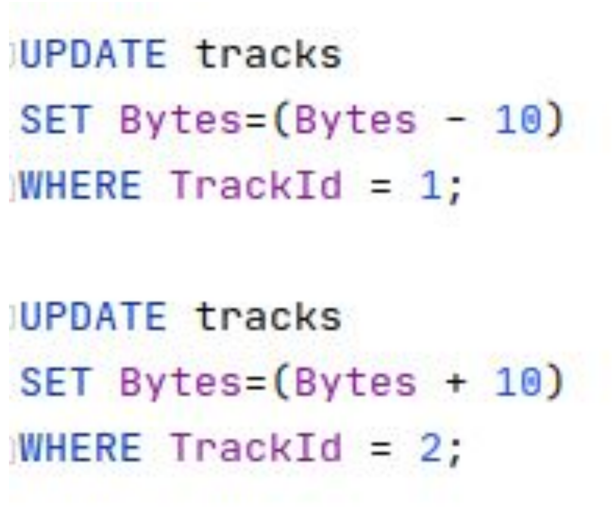
- A logical constraint on this operation is that the number of bytes in the tracks remains the same
    - By failing atomically we also fail from a consistency standpoint



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Independence

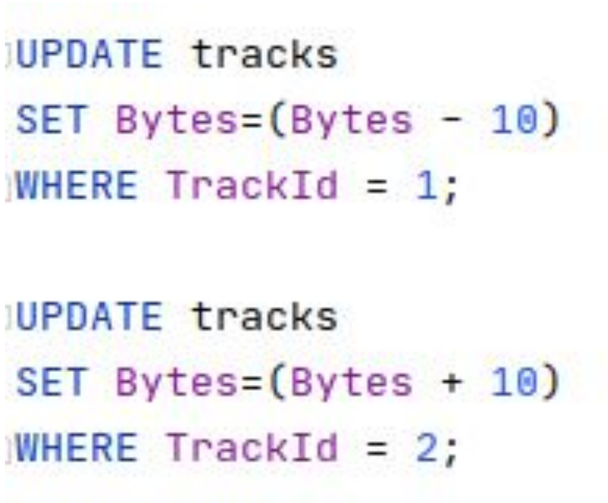
- Consider two users running this update at the same time
    - Both operations must complete as if the other operation was not concurrently running
    - E.g. User 2 cannot accidentally read the bytes value *before* User 1 and then write *after* User 1
        - We will discuss *Serializability* in a bit



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Durability

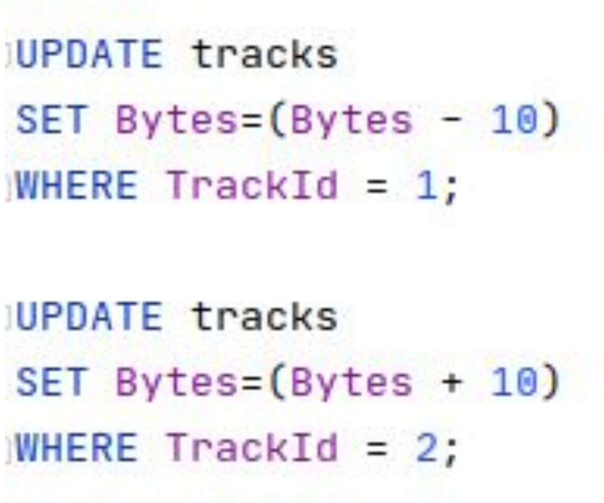
- If the system crashes *after* it has returned success for this transaction, when it restarts it *must* show the results of this transaction
  - This is often implemented with what is a *log file*



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

ACID Summary

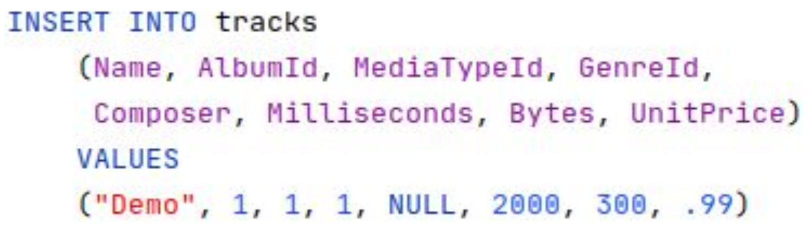
- ACID is a property that is *achievable* with databases but *not guaranteed*
  - ACID is *expensive* to guarantee
- You typically use *Transactions* to achieve some set of these properties



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Non-ACID Idea: Serializability

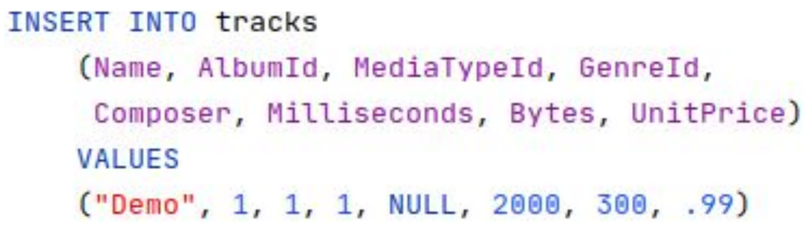
- Operations on your data must take place in a *logically serializable* order
- Consider two people trying to reserve the same seat on a plane
  - User 1 sees the open seat
  - User 2 sees the open seat
  - User 2 reserves the open seat
  - User 1 reserves the open seat



```
INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId,
     Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    ("Demo", 1, 1, 1, NULL, 2000, 300, .99)
```

Serializability

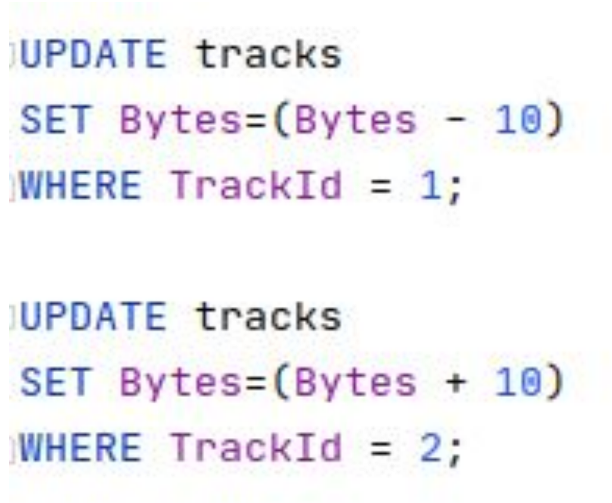
- Serializability: constraining this sequence of events to act as if each see/reserve action took place serially, rather than in parallel
- Many techniques for achieving serializability
  - Locking: the most common
- Related to Independence



```
INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId,
     Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    ("Demo", 1, 1, 1, NULL, 2000, 300, .99)
```

Transactions

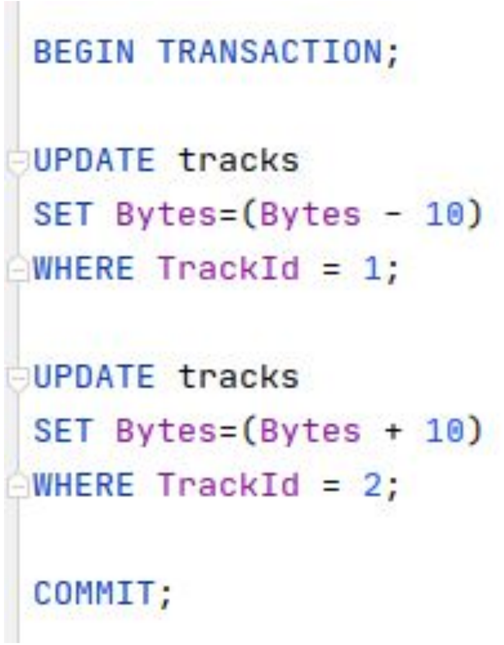
- Again, the DBMS solution to provide all of these properties are *Transactions*
- A transaction is a collection of operations that must be executed atomically
  - All succeed or all fail



```
UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;
```

Transactions

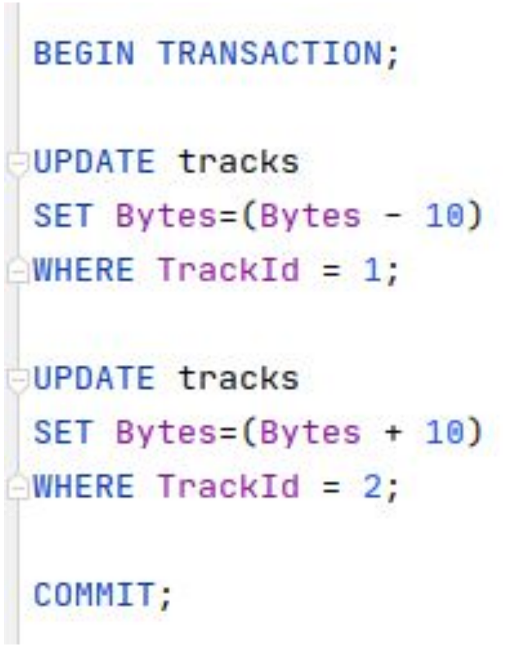
- Transaction syntax:
  - BEGIN TRANSACTION or START TRANSACTION to start a transaction
  - COMMIT to commit a transaction to the database
  - ROLLBACK to abort a transaction



```
BEGIN TRANSACTION;

UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;

COMMIT;
```

Transactions

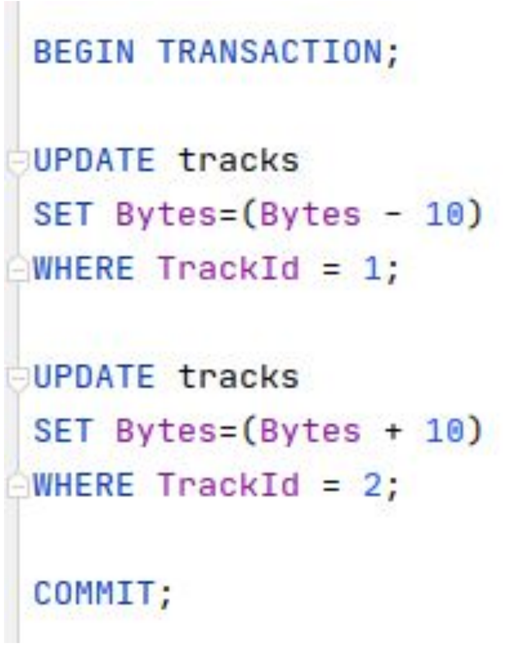
- By default, most DBMS are in *Auto-Commit* mode:
  When an SQL statement is not enclosed in a pair of start-transaction (BEGIN or SAVEPOINT) and end-transaction (COMMIT, ROLLBACK or RELEASE) SQL statements, it is executed in the database transaction implicitly delimited by the boundaries of the SQL statement. The SQL statement is said to be in auto-commit mode, since its database transaction is automatically delimited.



```
BEGIN TRANSACTION;

UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;

COMMIT;
```

Transactions

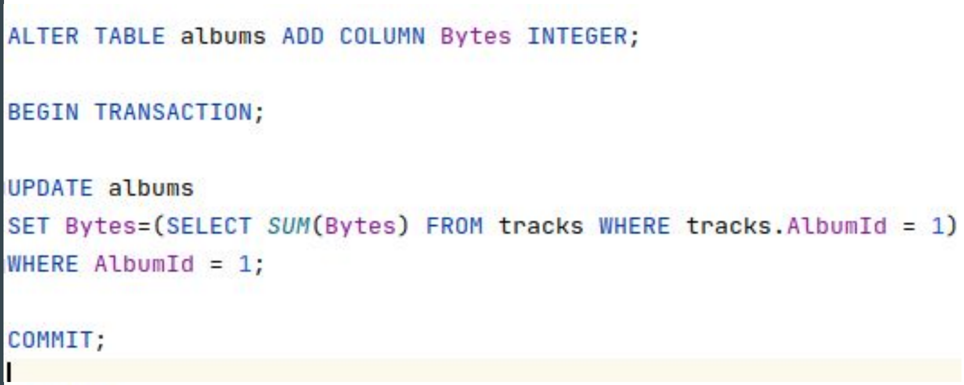
- SQLite operates in auto-commit mode when you interact with it in JDBC



```
BEGIN TRANSACTION;

UPDATE tracks
SET Bytes=(Bytes - 10)
WHERE TrackId = 1;

UPDATE tracks
SET Bytes=(Bytes + 10)
WHERE TrackId = 2;

COMMIT;
```

Isolation Levels

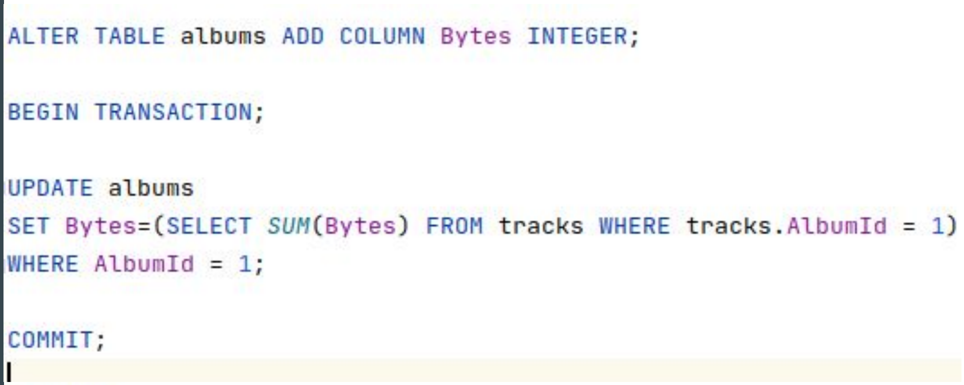
- Transactions can have different *isolation levels*
- Isolation levels determine how a transaction interacts with other concurrent transactions
- Allow you to pick what level of ACID you wish for your app
    - Trading off against *performance*



```sql
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels

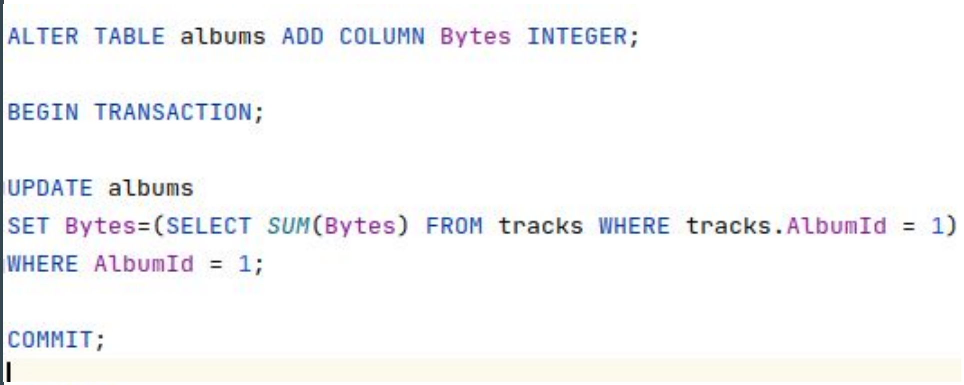
- *SERIALIZABLE*
  - Most restrictive isolation level
  - All reads and writes are locked
  - Additionally, and ranges scanned in a WHERE clause must be locked
    - This prevents *Phantom Reads* where data is missed because it is being inserted concurrently
  - Can be *very* expensive



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels

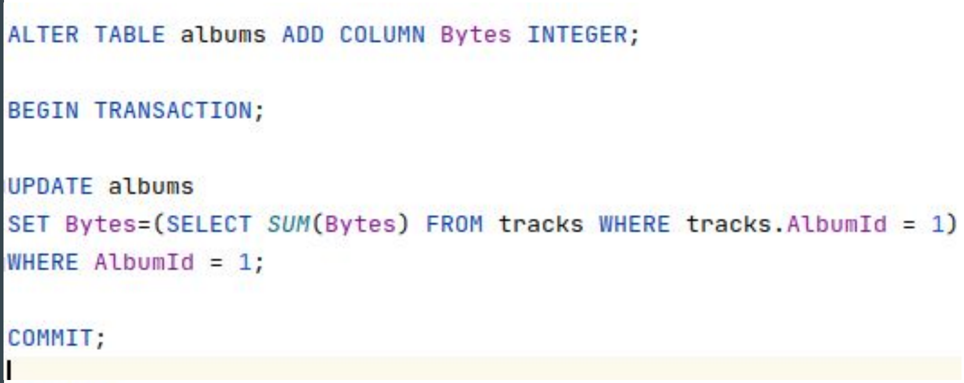
- *REPEATABLE READS*
  - All reads and writes are locked
  - No range locking
    - *Phantom Reads* can occur
  - Less expensive than *SERIALIZABLE*
  - Called repeatable reads because the data could be re-read at any point in the transaction and give the same results



```sql
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels

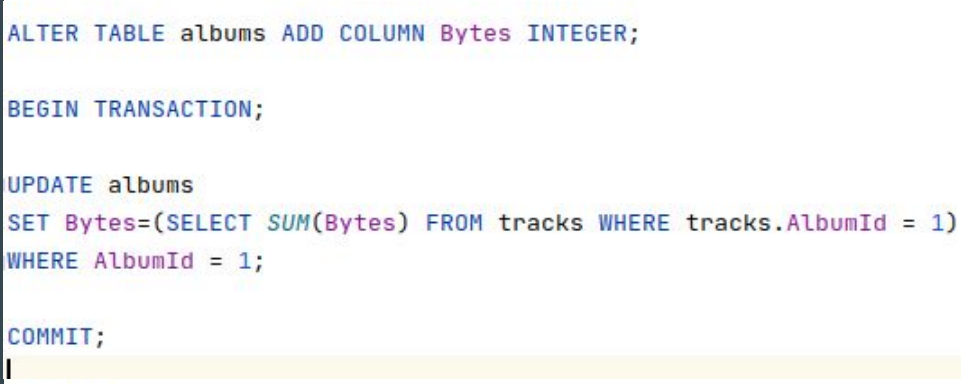
- *READ COMMITTED*
  - Writes are locked
  - Reads are only guaranteed to be part of a committed transaction
  - No range locking
    - *Phantom Reads* can occur
  - Less expensive than *REPEATABLE READS*



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels

- *READ UNCOMMITTED*
  - Writes are locked
  - Reads may be part of an *uncommitted* transaction
    - So called *Dirty Reads*
  - No range locking
    - *Phantom Reads* can occur
  - Least expensive, also the fewest guarantees

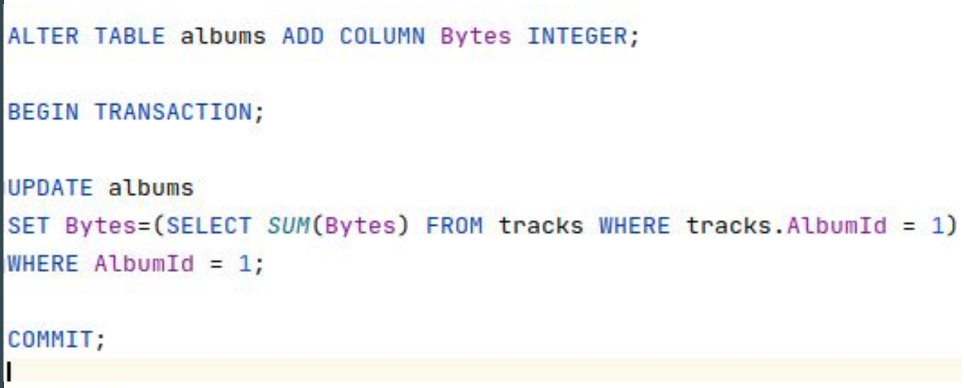


```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels

- Setting the transaction isolation level is different per database, but typically looks something like this

  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

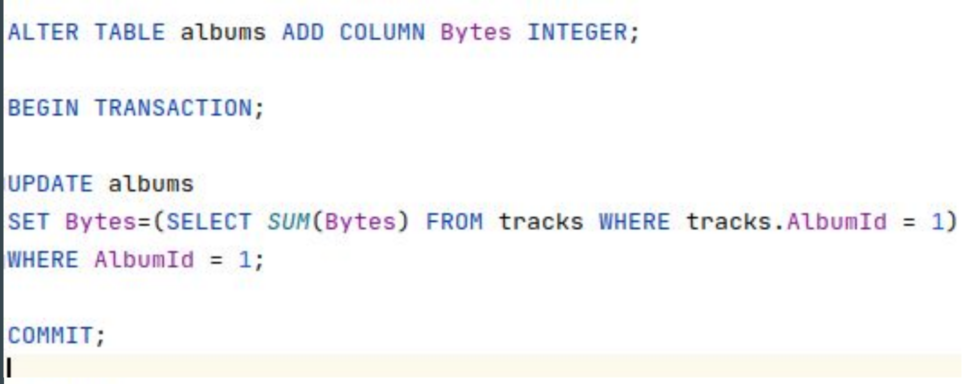


```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels in SQLite

- Transactions in SQLite are SERIALIZABLE

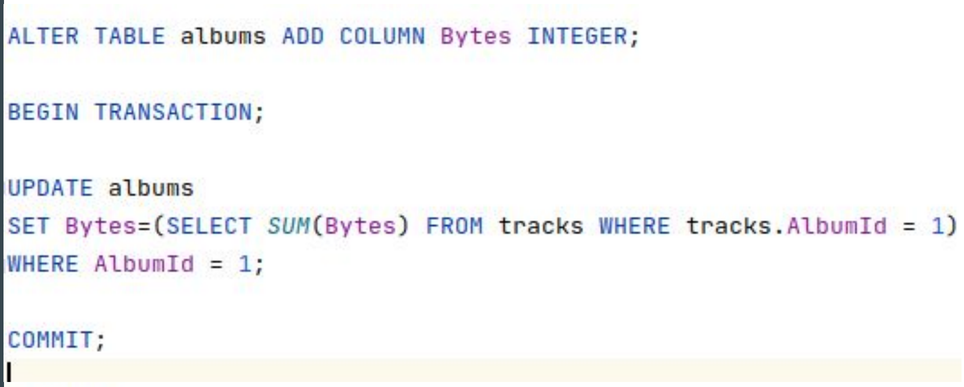
  "SQLite supports multiple simultaneous read transactions coming from separate database connections, possibly in separate threads or processes, but only one simultaneous write transaction."



```sql
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Isolation Levels in SQLite

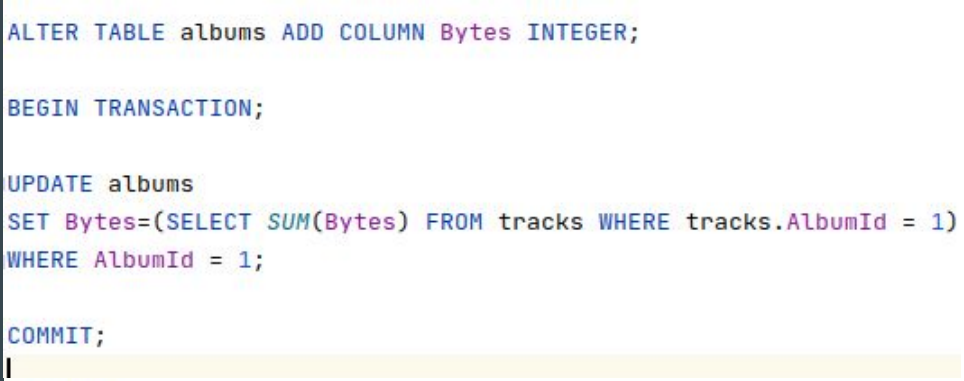
- This is *not* good for throughput in high-write applications
- Other databases are going to perform much better for high-write workloads



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Locking For Transactions

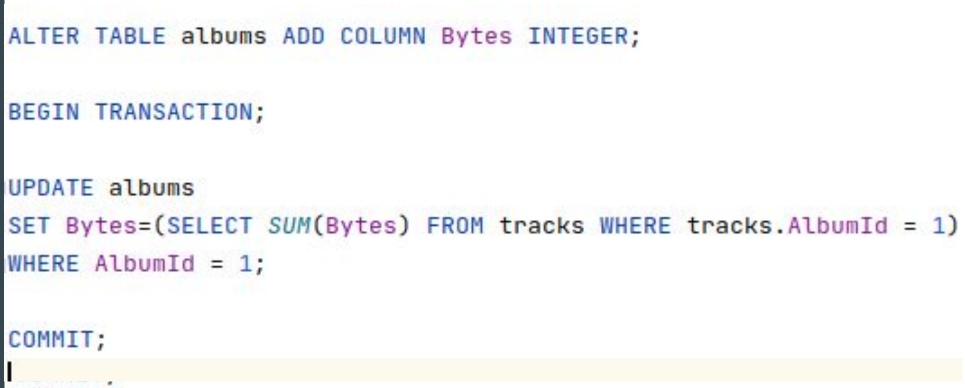
- To implement transactions, databases must implement specific locking strategies
- SERIALIZABLE must acquire *read locks* on all data (and ranges) read, as well as write locks on all updated data



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Lock Types

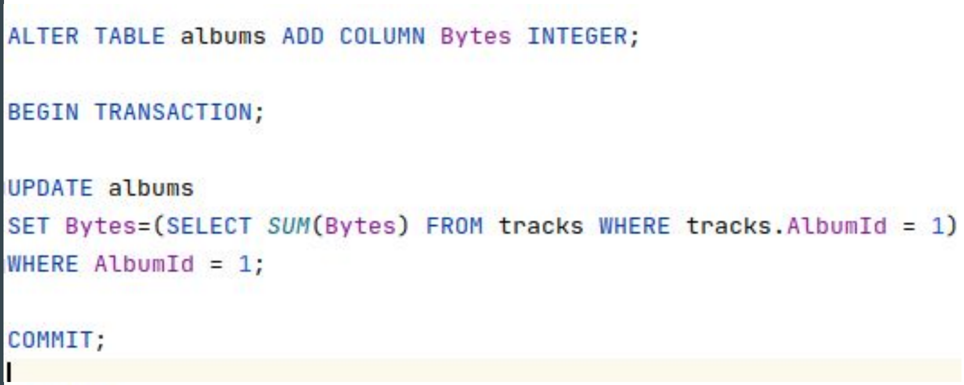
- Read locks can typically be held by multiple threads
  - We are all reading this data, and that's OK
- Write locks can only be held by one thread
  - Cannot be acquired if a read lock is already present on the data



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Lock Issues: Contention

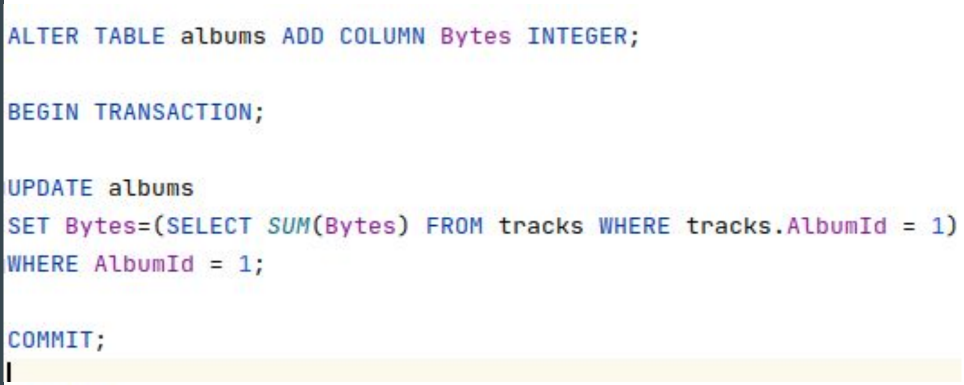
- Lock contention occurs when many database sessions all require frequent access to the same lock
  - Typically a write lock scenario
- If you have a “hot row” in your database that is updated frequently it can hurt throughput



```sql
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

Lock Issues: Blocking

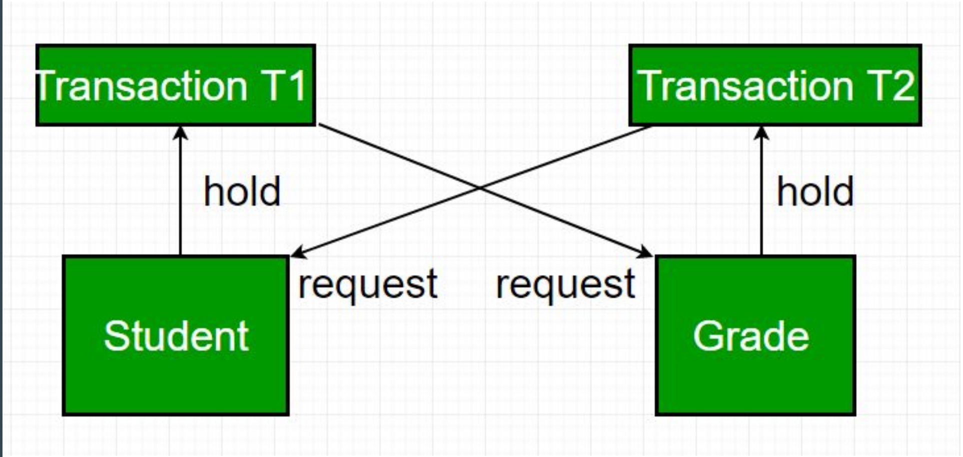
- Lock blocking occurs when a transaction holds a lock for a long period of time
- A long running batch process may, for example, cause your web site to lock up waiting for a read lock



```
ALTER TABLE albums ADD COLUMN Bytes INTEGER;

BEGIN TRANSACTION;

UPDATE albums
SET Bytes=(SELECT SUM(Bytes) FROM tracks WHERE tracks.AlbumId = 1)
WHERE AlbumId = 1;

COMMIT;
```

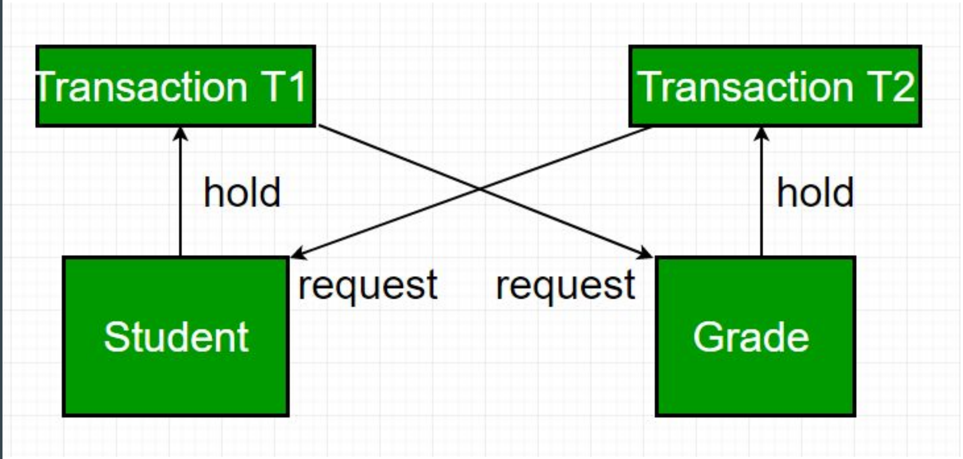
Lock Issues: Deadlock

- Deadlocks occur when two processes hold locks, and each needs the lock that the other holds
- Deadlocks are detected by some DBMS these days and one or the other transaction is aborted



Lock Issues: Deadlock

- To solve deadlocks, you can acquire locks in a specific order
    - E.g. acquire locks by table name



Transactions

- Transactions are used to guarantee data consistency
- Depending on your consistency needs and DBMS, there are varying levels of isolation available in transactions
- Transactions use locks to ensure data consistency
    - Issues arise from locking
        - Contention
        - Blocking
        - Deadlock

MONTANA
STATE UNIVERSITY