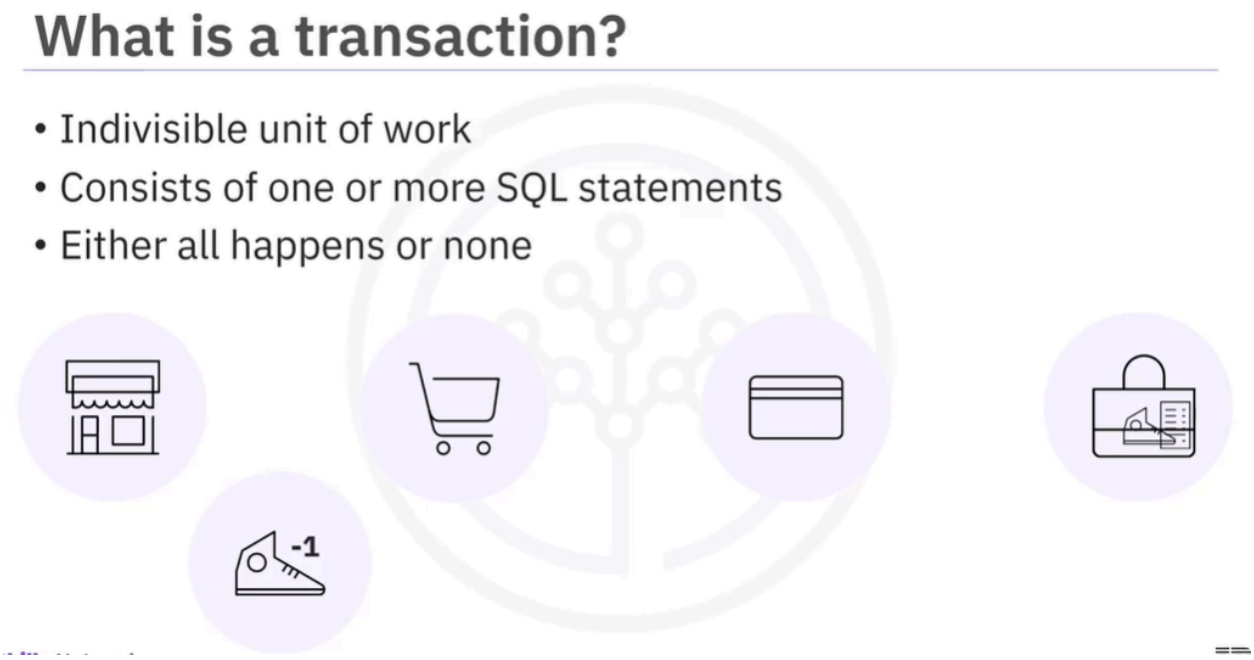
ACID



- A transaction is an indivisible unit of work. It can consist of one or more SQL statements, but to be considered successful, either all of those SQL statements must complete successfully, leaving the database in a new stable state, or none must complete, leaving the database as it was before the transaction began. For example, if you make a purchase using your bank card, many things must happen. The product must be added to your cart. Your payment must be processed. Your account must be debited the correct amount and the store's account credited. The inventory for that product must be reduced by the number purchased.

Transaction example

BankAccounts

AccountNumber	AccountName	Balance
B001	Rose	300
B002	James	13450
B003	Shoe Shop	124000
B004	Corner Shop	76000

ShoeShop

Product	Stock	Price
Boots	12	200
High heels	8	600
Brogues	10	150
Trainers	14	300

```
UPDATE BankAccounts
SET Balance = Balance-200
WHERE AccountName = 'Rose';
```

Transaction example

BankAccounts

AccountNumber	AccountName	Balance
B001	Rose	300
B002	James	13450
B003	Shoe Shop	124000
B004	Corner Shop	76000

ShoeShop

Product	Stock	Price
Boots	12	200
High heels	8	600
Brogues	10	150
Trainers	14	300

```
UPDATE BankAccounts
SET Balance = Balance+200
WHERE AccountName = 'Shoe Shop';
```

Transaction example

BankAccounts

AccountNumber	AccountName	Balance
B001	Rose	300
B002	James	13450
B003	Shoe Shop	124000
B004	Corner Shop	76000

ShoeShop

Product	Stock	Price
Boots	12	200
High heels	8	600
Brogues	10	150
Trainers	14	300

```
UPDATE ShoeShop
SET Stock = Stock-1
WHERE Product = 'Boots';
```

Transaction example

BankAccounts

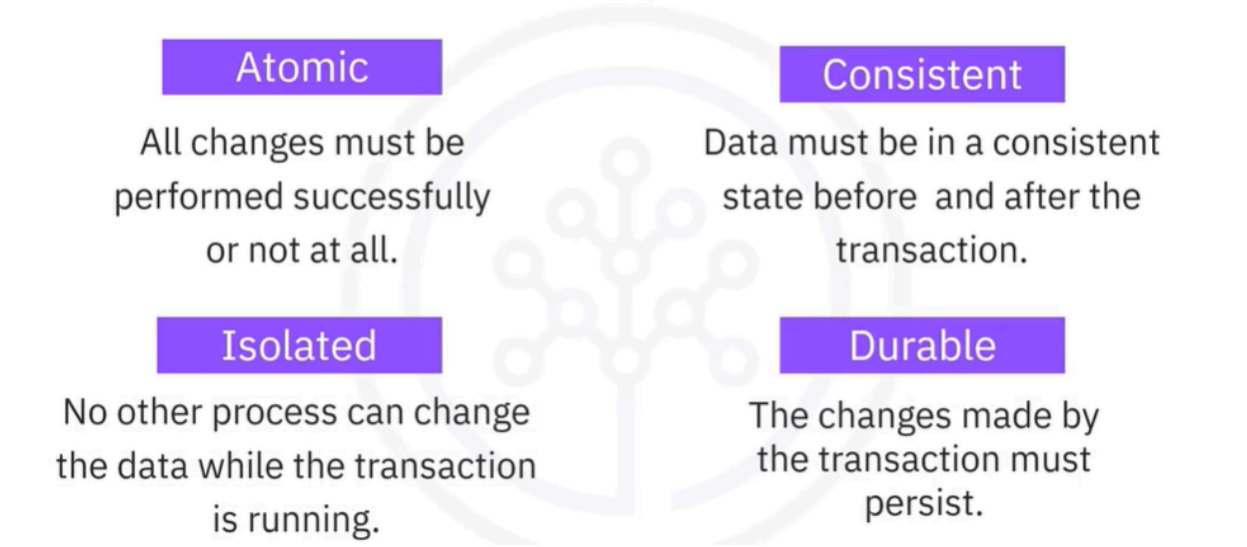
AccountNumber	AccountName	Balance
B001	Rose	300
B002	James	13450
B003	Shoe Shop	124000
B004	Corner Shop	76000

ShoeShop

Product	Stock	Price
Boots	12	200
High heels	8	600
Brogues	10	150
Trainers	14	300

If any of these UPDATE statements fail, the whole transaction must fail

- Let's look at the example in more detail. If Rose buys boots for $200, then you can use an UPDATE statement to decrease her account balance. And another UDATE statement to add $200 to the Shoe Shop balance. And a final update statement to decrease the stock level of boots at the Shoe Shop by 1. If any of these UPDATE statements fail, the whole transaction should fail, to keep the data in a consistent state.



- The types of transactions in the example are called ACID transactions. ACID stands for Atomic - All changes must be performed successfully or not at all. Consistent - Data must be in a consistent state before and after the transaction. Isolated - No other process can change the data while the transaction is running. Durable - The changes made by the transaction must persist.

ACID commands

- BEGIN
  - Start the ACID transaction
- COMMIT
  - All statements complete successfully
  - Save the new database state

```
BEGIN
        UPDATE BankAccounts
        SET Balance = Balance – 200
        WHERE AccountName = 'Rose';

        UPDATE BankAccounts
        SET Balance = Balance + 200
        WHERE AccountName = 'Shoe Shop';

        UPDATE ShoeShop
        SET Stock = Stock - 1
        WHERE Product = 'Boots';
COMMIT
```

ACID commands

- BEGIN
  - Start the ACID transaction
- COMMIT
  - All statements complete successfully
  - Save the new database state
- ROLLBACK
  - One or more statements fail
  - Undo changes

```
BEGIN
        UPDATE BankAccounts
        SET Balance = Balance – 200
        WHERE AccountName = 'Rose';

        UPDATE BankAccounts
        SET Balance = Balance + 200
        WHERE AccountName = 'Shoe Shop';

        UPDATE ShoeShop
        SET Stock = Stock - 1
        WHERE Product = 'Boots';
ROLLBACK
```

- To start an ACID transaction, use the command BEGIN. In db2 on Cloud, this command is implicit. Any commands you issue after that are part of the transaction, until you issue either COMMIT, or ROLLBACK. If all the commands complete successfully, issue a commit command to save everything in the database to a consistent, stable state. If any of the commands fail; perhaps Rose's account doesn't have enough money to make the payment, you can

issue a rollback command to undo all the changes and leave the database in its previously consistent stable state.

Calling ACID commands

- Can also be issued by some languages
  - Java, C, R, and Python
  - Requires the use of database specific APIs or connectors
- Use the EXEC SQL command to execute SQL statements from code

```
void main ()
  {
      EXEC SQL  UPDATE BankAccounts
        SET Balance = Balance - 200
        WHERE AccountName = 'Rose';
      EXEC SQL  UPDATE BankAccounts
        SET Balance = Balance + 200
        WHERE AccountName = 'Shoe Shop';
  EXEC SQL UPDATE ShoeShop
        SET Stock = Stock - 1
        WHERE Product = 'Boots';
      FINISHED:
      EXEC SQL  COMMIT WORK;
      return;

   SQLERR:
     EXEC SQL  WHENEVER SQLERROR CONTINUE;
     EXEC SQL  ROLLBACK WORK;
     return;
}
```

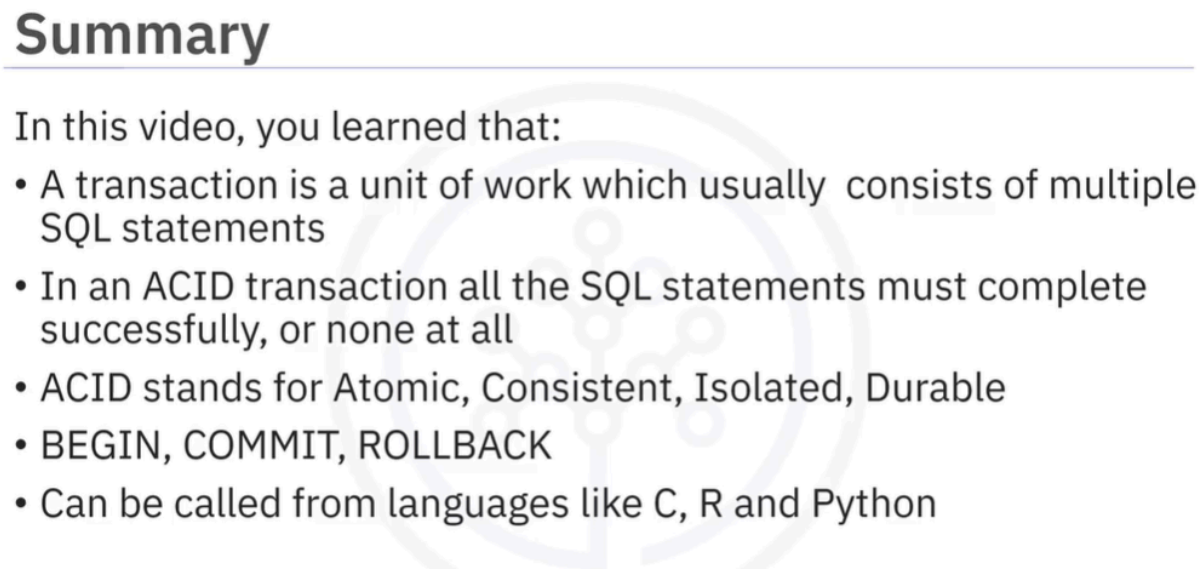
Calling ACID commands

- Can also be issued by some languages
  - Java, C, R, and Python
  - Requires the use of database specific APIs or connectors
- Use the EXEC SQL command to execute SQL statements from code

```
void main ()
  {
      EXEC SQL  UPDATE BankAccounts
        SET Balance = Balance - 200
        WHERE AccountName = 'Rose';
      EXEC SQL  UPDATE BankAccounts
        SET Balance = Balance + 200
        WHERE AccountName = 'Shoe Shop';
  EXEC SQL UPDATE ShoeShop
        SET Stock = Stock - 1
        WHERE Product = 'Boots';
      FINISHED:
      EXEC SQL  COMMIT WORK;
      return;

   SQLERR:
     EXEC SQL  WHENEVER SQLERROR CONTINUE;
     EXEC SQL  ROLLBACK WORK;
     return;
}
```

- SQL statements can be called from languages like Java, C, R, and Python. This requires the use of database-specific access APIs such as Java Database Connectivity (JDBC) for Java or a specific database connector like ibm_db for Python. Most languages use the EXEC SQL commands to initiate a SQL command, including COMMIT and ROLLBACK, as you can see in this example. Remember that BEGIN is implicit, you do not need to call it out explicitly. Incorporating SQL commands into your application code gives you the opportunity to create error-checking routines that in turn control whether the transaction is committed or rolled back.



ACID Transactions: A Simple Explanation

In the world of databases, an ACID transaction is like a promise that ensures everything happens perfectly when you make a change. Imagine you're baking a cake. You need to mix the ingredients, bake it, and then frost it. If you forget to add sugar, the cake will be ruined! Similarly, in an ACID transaction, all parts of a task must be completed successfully, or none of them will happen at all. This way, the database remains in a stable and consistent state, just like your cake should be delicious and well-made.

Let’s say you’re buying a pair of boots online. When you click "buy," several things need to happen: your account is charged, the store gets the money, and the inventory is updated. If any of these steps fail—like if your account doesn’t have enough money—the whole transaction is canceled, and everything goes back to how it was before. This is done using commands like BEGIN to start the transaction, COMMIT to save the changes if everything goes well, and ROLLBACK to undo everything if something goes wrong.

How can you apply ACID principles in programming?

To apply ACID principles in programming, especially when working with databases, you can follow these guidelines:

1. **Atomicity**:
    - Ensure that a transaction is treated as a single unit. Use commands that group multiple operations together. If one operation fails, the entire transaction should fail.
    - Example: In SQL, you can use `BEGIN`, `COMMIT`, and `ROLLBACK` to manage transactions.
2. **Consistency**:
    - Make sure that any transaction brings the database from one valid state to another. This means enforcing rules and constraints (like foreign keys) to maintain data integrity.
    - Example: Before committing a transaction, check that all data adheres to the defined rules.
3. **Isolation**:
    - Ensure that transactions do not interfere with each other. Use locking mechanisms or isolation levels to prevent other transactions from accessing data being modified until the transaction is complete.

- Example: In SQL, you can set the isolation level using commands like `SET TRANSACTION ISOLATION LEVEL`.

4. **Durability**:
    - Once a transaction is committed, the changes should persist even in the event of a system failure. This often involves using database features that log transactions.
    - Example: Most databases automatically handle durability by writing changes to a transaction log.

By following these principles, you can ensure that your database operations are reliable and maintain data integrity.

How can you implement ACID transactions in your applications?

To implement **ACID transactions** in your applications, you can follow these steps:

1. **Use a Database Management System (DBMS)**:
    - Choose a DBMS that supports ACID transactions, such as MySQL, PostgreSQL, or Oracle. Ensure that your database is configured to handle transactions.
2. **Begin a Transaction**:
    - Start a transaction using the appropriate command. In SQL, you can use `BEGIN` to indicate the start of a transaction.
    - Example:

```
BEGIN;
```

3. **Perform Database Operations**:
    - Execute the necessary SQL statements (e.g., `INSERT`, `UPDATE`, `DELETE`) as part of the transaction. Ensure that these operations are logically grouped

together.

- Example:

```
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
```

4. **Check for Errors**:
   - After executing the operations, check for any errors or conditions that may require rolling back the transaction. This can include validation checks or business logic conditions.
5. **Commit or Rollback**:
   - If all operations are successful, use the `COMMIT` command to save the changes to the database. If any operation fails, use the `ROLLBACK` command to undo all changes made during the transaction.
   - Example:

```
COMMIT;  -- To save changes
-- or
ROLLBACK;  -- To undo changes
```

6. **Use Programming Language APIs**:
   - If you are using a programming language (like Python, Java, or C#), utilize the database-specific APIs or libraries that support transactions. For example, in Python, you can use the `sqlite3` or `SQLAlchemy` libraries to manage transactions.
   - Example in Python:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
try:
    cursor.execute("BEGIN;")
    cursor.execute("UPDATE accounts SET balance = balance - 100 WH
ERE account_id = 1;")
    cursor.execute("UPDATE accounts SET balance = balance + 100 W
HERE account_id = 2;")
    conn.commit()  # Save changes
except Exception as e:
    conn.rollback()  # Undo changes
finally:
    conn.close()
```

By following these steps, you can effectively implement ACID transactions in your applications, ensuring data integrity and reliability.

How can you ensure Consistency in a transaction?

To ensure **Consistency** in a transaction, you can follow these practices:

1. **Define Constraints**:
    - Use database constraints such as **primary keys**, **foreign keys**, **unique constraints**, and **check constraints** to enforce rules on the data.
    - Example: A foreign key constraint ensures that a value in one table must exist in another table, maintaining referential integrity.
2. **Use Transactions**:
    - Always wrap your operations in a transaction. This ensures that all changes are applied together, and if any part fails, the entire transaction can be rolled back.
    - Example: In SQL, use `BEGIN`, `COMMIT`, and `ROLLBACK` to manage transactions.
3. **Validate Data Before Committing**:

- Before committing a transaction, validate that all data meets the required conditions and business rules.
- Example: Check that an account has sufficient funds before processing a payment.

4. **Implement Business Logic**:
    - Ensure that your application logic enforces the rules of your business domain. This can include checks for valid states or conditions before making changes.
    - Example: If a product is out of stock, prevent the transaction from proceeding.
5. **Use Stored Procedures**:
    - Encapsulate your transaction logic in stored procedures to ensure that all operations are executed in a controlled manner.
    - Example: A stored procedure can handle the entire process of placing an order, including checks and updates.

By implementing these practices, you can maintain consistency in your transactions and ensure that your database remains in a valid state.

What would happen if Consistency is not maintained in transactions?

If **Consistency** is not maintained in transactions, several issues can arise:

1. **Data Integrity Issues**:
    - The database may contain invalid or contradictory data. For example, if a transaction allows a product to be sold even when it is out of stock, it can lead to discrepancies in inventory records.
2. **Inconsistent States**:

- The database may end up in an inconsistent state where some data reflects the changes made by a transaction while other data does not. This can lead to confusion and errors in data retrieval and reporting.

3. **Violation of Business Rules**:
    - Business logic may be compromised. For instance, if a financial transaction allows an account to go into a negative balance, it violates the business rule that accounts should not be overdrawn.
4. **Complicated Error Recovery**:
    - If consistency is not enforced, recovering from errors becomes more complex. You may need to manually identify and correct inconsistencies, which can be time-consuming and error-prone.
5. **Loss of Trust**:
    - Users and stakeholders may lose trust in the system if they encounter inconsistent or incorrect data. This can lead to a lack of confidence in the application and its reliability.

Maintaining consistency is crucial for ensuring that the database accurately reflects the real-world scenarios it is meant to model.

What would be the impact of inconsistent data on business operations?

The impact of inconsistent data on business operations can be significant and may include:

1. **Poor Decision-Making**:
    - Inconsistent data can lead to incorrect insights and analyses, resulting in misguided business decisions. For example, if sales data is inaccurate, management may misjudge market trends or customer demand.
2. **Operational Inefficiencies**:

- Teams may spend extra time reconciling data discrepancies, leading to wasted resources and reduced productivity. This can slow down processes and hinder timely responses to market changes.

3. **Customer Dissatisfaction**:
    - Inconsistent data can result in errors in customer orders, billing, or service delivery. This can lead to frustration and dissatisfaction among customers, damaging the company's reputation and customer loyalty.
4. **Compliance Risks**:
    - Many industries are subject to regulations that require accurate reporting and data management. Inconsistent data can lead to compliance violations, resulting in legal penalties and financial losses.
5. **Increased Costs**:
    - The need to correct errors and manage inconsistencies can lead to increased operational costs. This includes costs associated with data cleaning, additional staff time, and potential loss of revenue due to errors.
6. **Loss of Competitive Advantage**:
    - Companies that rely on accurate data for strategic planning may find themselves at a disadvantage compared to competitors who maintain consistent and reliable data.

Ensuring data consistency is essential for smooth business operations and maintaining a competitive edge.

What would happen if a transaction is not atomic?

If a transaction is not **atomic**, several issues can arise:

1. **Partial Updates**:
    - Some operations within the transaction may succeed while others fail, leading to a situation where the database is only partially updated. This

can result in inconsistent data states.

2. **Data Integrity Violations**:
    - The integrity of the data may be compromised. For example, if a transaction involves transferring funds between accounts and only one account is updated, it can lead to discrepancies in account balances.
3. **Inconsistent Application State**:
    - The application may operate on inconsistent data, leading to incorrect behavior or errors. For instance, if a user sees outdated information due to a partial transaction, it can cause confusion.
4. **Complicated Error Recovery**:
    - Recovering from a non-atomic transaction can be complex and time-consuming. You may need to manually identify and correct inconsistencies, which increases the risk of human error.
5. **Loss of Trust**:
    - Users may lose trust in the system if they encounter issues stemming from non-atomic transactions. This can damage the reputation of the application and lead to customer dissatisfaction.
6. **Increased Complexity in Business Logic**:
    - Developers may need to implement additional checks and error-handling mechanisms to manage the consequences of non-atomic transactions, increasing the complexity of the codebase.

Ensuring atomicity is crucial for maintaining data integrity and reliability in database operations.

How can you ensure atomicity in transactions?

To ensure **atomicity** in transactions, you can follow these practices:

1. **Use Transaction Management**:

- Utilize the transaction management features provided by your database management system (DBMS). This typically involves using commands like `BEGIN`, `COMMIT`, and `ROLLBACK`.

2. **Wrap Operations in a Transaction Block**:
   - Group all related operations within a transaction block. This ensures that either all operations are completed successfully, or none are applied.
   - Example:

```
BEGIN;
-- Perform multiple operations
COMMIT;  -- Save changes if all operations succeed
```

3. **Error Handling**:
   - Implement error handling in your application code to catch exceptions or errors during transaction execution. If an error occurs, roll back the transaction to maintain atomicity.
   - Example in Python:

```
try:
    # Start transaction
    cursor.execute("BEGIN;")
    # Perform operations
    cursor.execute("UPDATE accounts SET balance = balance - 100 WH
ERE account_id = 1;")
    cursor.execute("UPDATE accounts SET balance = balance + 100 W
HERE account_id = 2;")
    # Commit if successful
    conn.commit()
except Exception as e:
    # Rollback on error
    conn.rollback()
```

4. **Use Stored Procedures**:

- Encapsulate your transaction logic in stored procedures. This allows you to manage the transaction as a single unit of work, ensuring atomicity.
- Example:

```
CREATE PROCEDURE TransferFunds(@fromAccount INT, @toAccount INT, @amount DECIMAL)
AS
BEGIN
   BEGIN TRANSACTION;
   -- Perform operations
   COMMIT;  -- Commit if successful
END;
```

5. **Database Isolation Levels**:
   - Set appropriate isolation levels in your database to control how transactions interact with each other. This can help prevent issues that may arise from concurrent transactions.
   - Example: Use `SERIALIZABLE` isolation level for strict atomicity.

By implementing these practices, you can effectively ensure atomicity in your transactions, maintaining data integrity and reliability.

How could you apply atomicity in a real-world scenario?

Applying **atomicity** in a real-world scenario can be illustrated through a banking transaction, such as transferring funds between two accounts. Here's how you can implement atomicity in this context:

Scenario: Transferring Funds Between Accounts

1. **Transaction Start**:

- Begin the transaction to ensure that all operations are treated as a single unit of work.

```
BEGIN;
```

2. **Check Account Balances**:
   - Verify that the source account has sufficient funds to cover the transfer amount. If not, roll back the transaction.

```
SELECT balance FROM accounts WHERE account_id = 1;  -- Source accou
nt
```

3. **Update Source Account**:
   - Deduct the transfer amount from the source account.

```
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;  --
Deduct $100
```

4. **Update Destination Account**:
   - Add the transfer amount to the destination account.

```
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;  -
- Add $100
```

5. **Commit or Rollback**:
   - If all operations succeed, commit the transaction to save the changes. If any operation fails (e.g., insufficient funds), roll back the transaction to maintain atomicity.

```
COMMIT;  -- Save changes if all operations succeed
-- or
ROLLBACK;  -- Undo changes if any operation fails
```

Example in Python:

Here's how you might implement this in Python using a database connection:

```
import sqlite3

conn = sqlite3.connect('bank.db')
cursor = conn.cursor()

try:
    cursor.execute("BEGIN;")

    # Check balance
    cursor.execute("SELECT balance FROM accounts WHERE account_id = 1;")
    balance = cursor.fetchone()[0]

    if balance >= 100:
        # Deduct from source account
        cursor.execute("UPDATE accounts SET balance = balance - 100 WHERE
account_id = 1;")
        # Add to destination account
        cursor.execute("UPDATE accounts SET balance = balance + 100 WHERE
account_id = 2;")
        # Commit the transaction
        conn.commit()
    else:
        raise Exception("Insufficient funds")
except Exception as e:
    conn.rollback()  # Rollback on error
finally:
    conn.close()
```

Key Points:

- **Atomicity** ensures that either all operations (deducting from one account and adding to another) are completed successfully, or none are applied, preventing any inconsistencies in account balances.

- This approach maintains data integrity and provides a reliable banking experience for users.