**DATA DEFINITION LANGUAGE (DDL)**

**First steps in every database creation, as they define the objects that will contain and manage the data and server-side code**

|  |  |
| --- | --- |
| **Statement** | **Object** |
| CREATE  ALTER  DROP | Database  Schema  Table  View  Index  Type  Procedure  Function  Role  Trigger  Rule  Sequence |

**DDL Examples:**

**Create database:**

CREATE DATABASE DML;

**Create table:**

CREATE TABLE Orders (OrderID INT PRIMARY KEY,

OrderDate Date NOT NULL);

**Add a column to our table:**

ALTER TABLE Orders ADD Customer VARCHAR(10);

**Drop the database and everything that was in it:**

DROP DATABASE DML;

**DATA CONTRTOL LANGUAGE (DCL)**

**Manages security, what users can and can’t do on objects in the database**

|  |  |  |  |
| --- | --- | --- | --- |
| **Statement** | **Permission** | **Object** | **Role** |
| GRANT  REVOKE | SELECT  INSERT  UPDATE  DELETE  CREATE  ALL | **ON**  *(Object)* | **TO**  **FROM**  *(Role)*  PUBLIC  CURRENT\_USER  SESSION\_USER |

**DCL Examples:**

**To allow the Accounting role to retrieve data from the Customers table or view:**

GRANT SELECT ON Customers TO Accounting

**To disallow Deleting orders from all known owners:**

REVOKE DELETE ON Orders FROM Public

**Granting READ permissions to the Public for the Orders table:**

GRANT SELECT ON Orders TO PUBLIC;

**Granting all possible permissions on the Orders table to a custom Administrators role:**

GRANT ALL ON Orders TO Administrators;

**Revoking all permissions from all known owners for the Orders table:**

REVOKE ALL ON Orders FROM PUBLIC;

**DATA MANIPULATION LANGUAGE (DML):**

**INSERT**

**Used to load data into tables**

**UPDATE**

**Used to modify existing data**

**DELETE**

**Used to delete existing data**

**DML Examples**

**To insert order #25 using row, value constructor:**

INSERT INTO Orders (OrderId, OrderDate, Customer)

VALUES (25, ‘2019-01-01’, ‘John’);

**To update the customer name in the Orders table to null for customers who do not appear in the Customers table:**

UPDATE Orders

SET Customers = NULL

WHERE Customer NOT IN (SELECT Customer FROM Customers);

**DATA QUERY LANGUAGE (DQL)**

**Used to return data to the client application**

**SELECT <Expressions>**

**FROM <Table Source>**

**WHERE <Filter>**

**PostgreSQL also allows for the “returning clause” which allows us to process values that were generated by the DML or archive the rows that were affected by it**

**DML Examples:**

**To return the entire Orders table:**

SELECT \* FROM Orders;

**To return the row set from an UPDATE statement that will not only modify the Customers to NULL, but will return all Customer rows for the orders that were updated:**

UPDATE Orders

SET Customer = NULL

WHERE Customer NOT IN (SELECT Customer FROM Customers)

RETURNING \*;

**DATA TRANSACTION LANGUAGE (DTL)**

**A transaction block consists of a**

**Starting block:**

**[BEGIN | START] [WORK | TRANSACTION]**

**Potential save points during the transaction**

**SAVEPOINT <Name>**

**Either a commit or a rollback that ends a transaction and defines its second boundary**

**COMMIT [WORK | TRANSACTION]**

**[TO SAVEPOINT <Name>]**

**ROLLBACK [WORK | TRANSACTION]**

**DTL Examples:**

**Begins the transaction:**

BEGIN TRANSACTION;

**Inserts or updates or deletes something:**

INSERT <Something>;

UPDATE <Something>;

DELETE <Something>;

**Checks whether or not some condition is met, and either rollback or commit the transaction**

IF <Condition>

ROLLBACK TRANSACTION

ELSE COMMIT TRANSACTION;

END IF;

**Programming Paradigms**

Classify programming languages based on its characteristics

* **Declarative**—the programmer declares properties of the desired result, but not how to compute it
  + --SQL
  + --tells it WHAT to do
* **Imperative**—the programmer instructs the machine how to change its state
  + --tells the engine HOW we want things to be done
  + PostgreSQL use imperative constructs for
    - Flow control: IF, LOOP, WHILE, FOR, CONTINUE, EXIT, GOTO
    - Variables : DECLARE, <Variable> <Type> [:=<Expression>]
    - Error handling: RAISE NOTICE, WHENEVER, EXCEPTION
    - Management: SET, Backups, Replication, VACUUM, Statistics

**CONCURRENCY CONFLICTS**

**Occur when two or more processes or any type of resource consumers compete for shared resources**

**Database Transaction**: A transaction symbolizes a unit of work performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions

* Database ACID Properties:
  + **Atomicity**: indivisible and irreducible series of database operations
    - A set of operations can be bounded as a single unit of work (transaction) which either succeeds or fails as a whole
    - Example: an ATM transaction bound must include all of three transactions: (1) check account balance, (2) debit account, (3) issue cash
  + **Consistency**: changes to the database allowed from one valid state to another
    - Any transaction can only change the state of the database from one valid state to another valid state
    - Example: the database must not allow a transaction between a consistent state and an inconsistent state
  + **Isolation**: when and how changes become visible to others
    - Determines how different transactions are kept isolated from one another
    - Example: If both Transaction 1 and Transaction 2 modify A, B, and C, then it must be determined somewhere which modification takes precedence so that we can know if Transaction 2, for instance, will acknowledge the changes that have been made in Transaction 1
  + **Durability**: committed changes survive permanently
    - When a transaction has been acknowledged as committed to the database, it must withstand any system failure no matter if it happens a millisecond after committing and acknowledging the work to the client
    - Not as obvious b/c changes are made in RAM and it could blow up, so changes must be saved in a non-volatile medium such as a disk

**Isolation Paradigms**

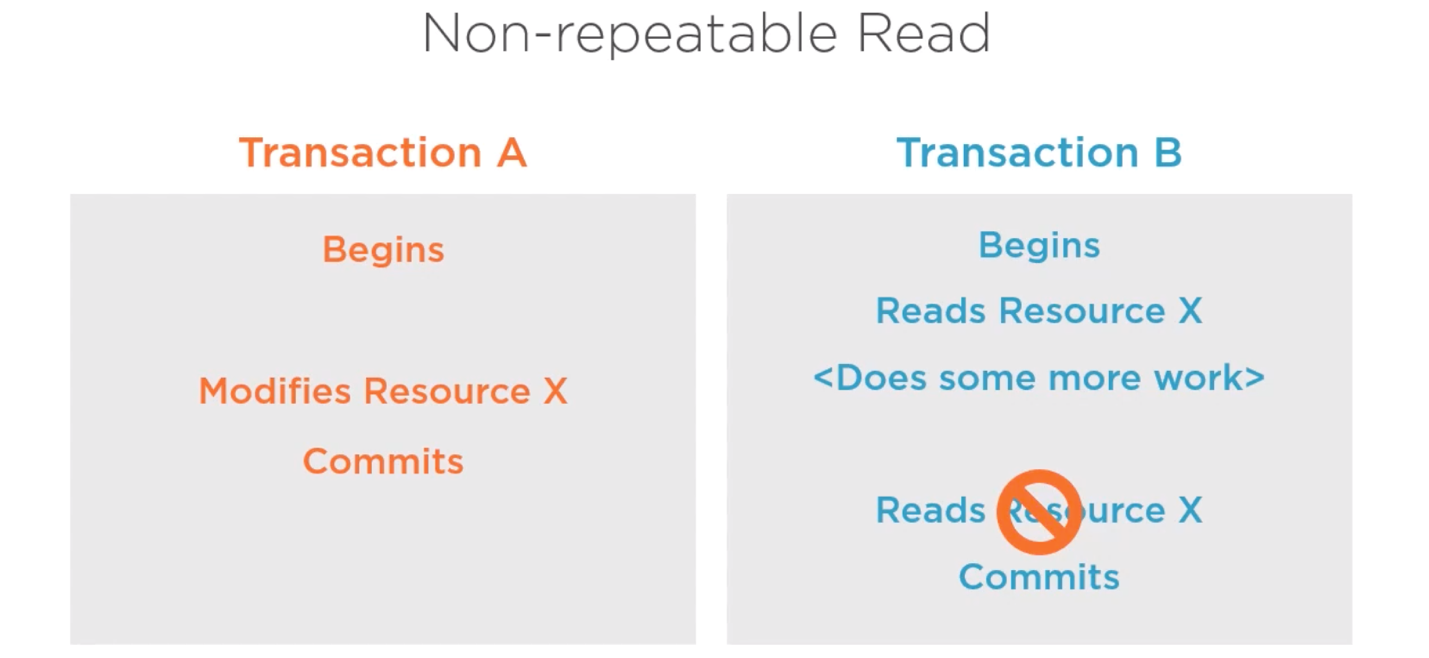
* **Pessimistic**: this approach assumes that there will be conflict so it takes measures to prevent it in advance
  + Uses locks
  + Introduces waits (blocks)
  + Uses a single copy of data
* **Optimistic**: assumes that conflicts will be rare, but requires a validity check before committing
  + Uses multiple snapshots/copies of the data
  + May result in aborts/retries
  + Costly in memory and disk
  + PostgreSQL uses a MultiVersion Concurrency Control (MVCC) System

**Understanding ANSI (American National Standard Institute) Concurrency Phenomena**

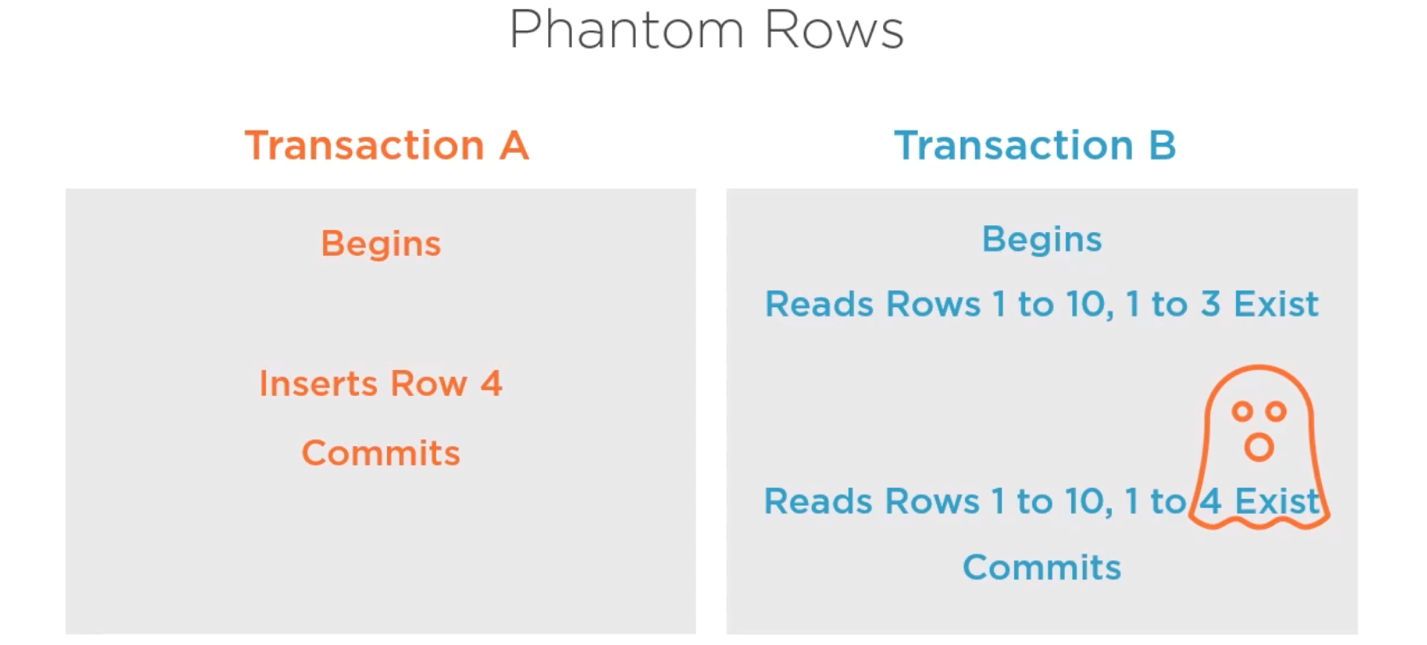
1. **Dirty Read**
   1. Occurs when a transaction reads a resource that has been modified by a concurrent transaction before it commits, but the modifying transaction decides to road back its changes. Even though the transaction may have read in the resource and acted on it, it’s like the resource modification had never occurred in the first place.
   2. A transaction reads a state that was never committed
   3. This doesn’t happen in PostgreSQL, but does in MySQL, SQL Server, etc.



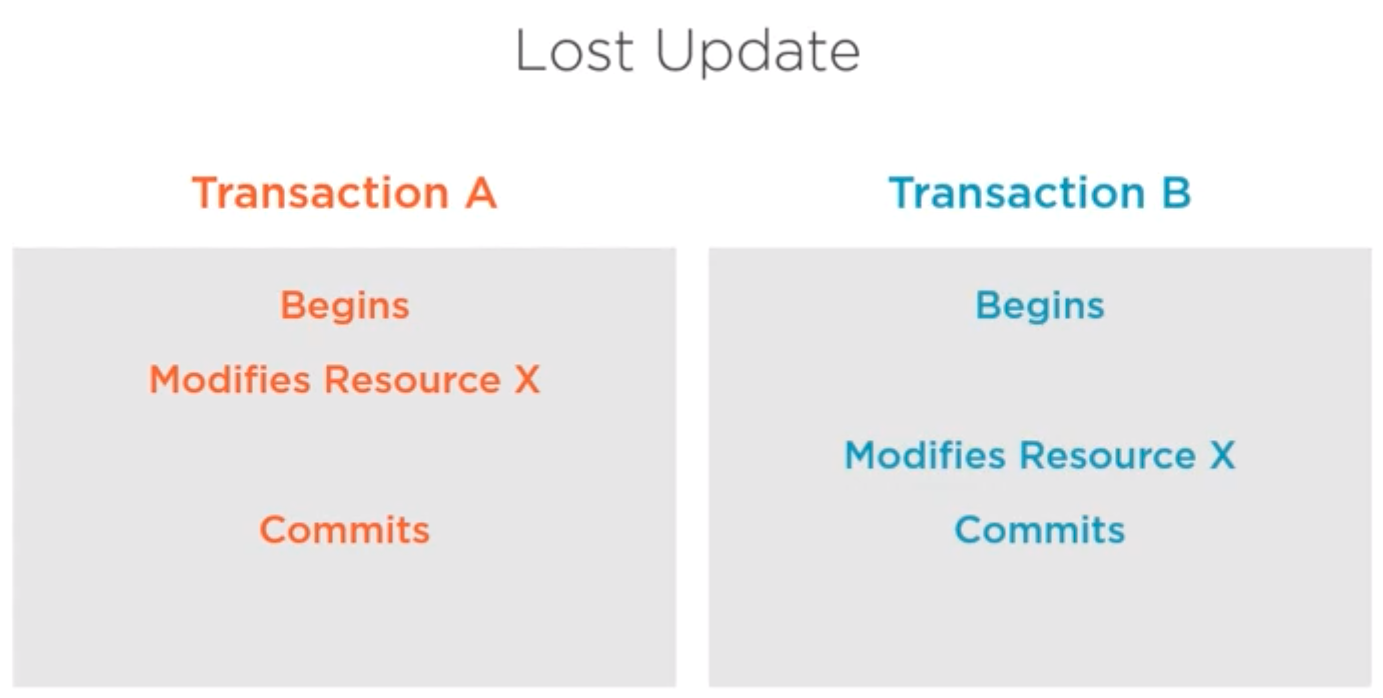
1. **Non-repeatable Read**
   1. Occurs when a transaction is unable to read a resource multiple times due to concurrent changes made by multiple transactions
   2. The same query executed twice or more returns inconsistent results for the same rows



1. **Phantom Rows**
   1. Rows appear out of nowhere



1. **Lost Update**
   1. Transaction A updates something, Transaction B does something, essentially overwriting the modifications Transaction A did. When Transaction A and Transaction B tries to commit the changes, Transaction A doesn’t know that it has been overwritten and that work is lost. Transaction B was the latest transaction so its work is saved in the commit.
   2. This doesn’t happen in PostgreSQL or most relational database engines

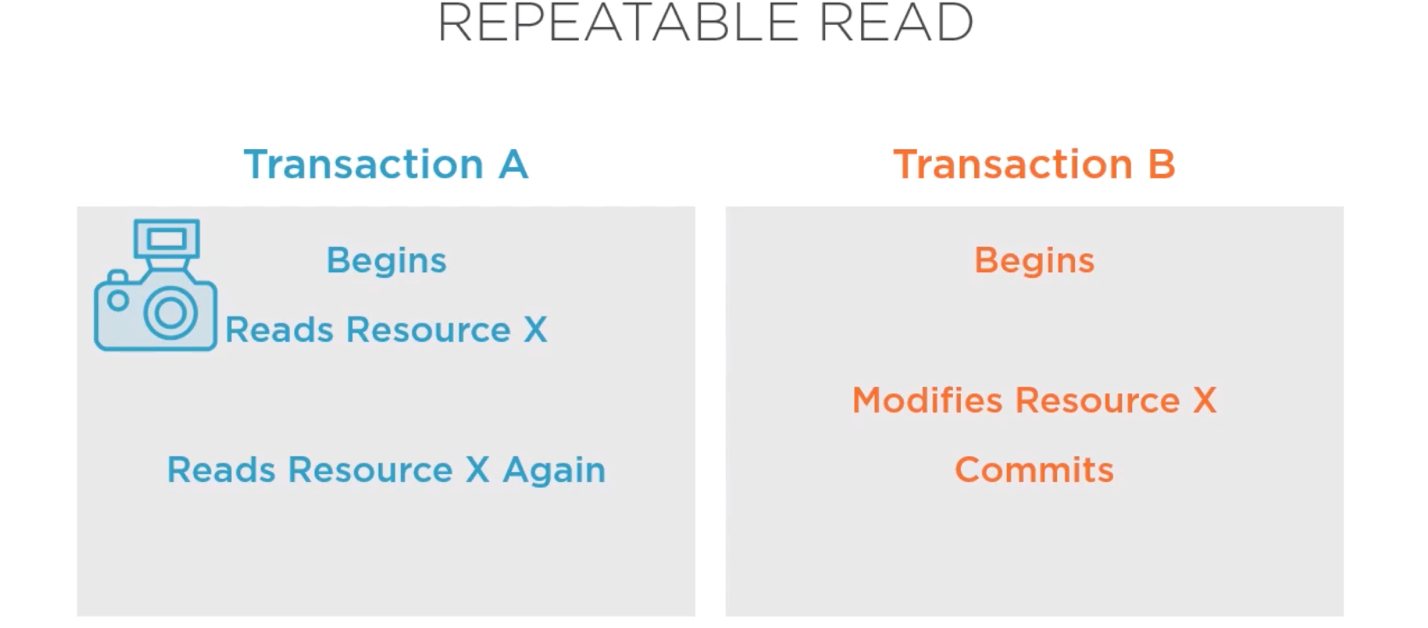


**How ANSI Isolation Levels Solve Concurrency Phenomena**

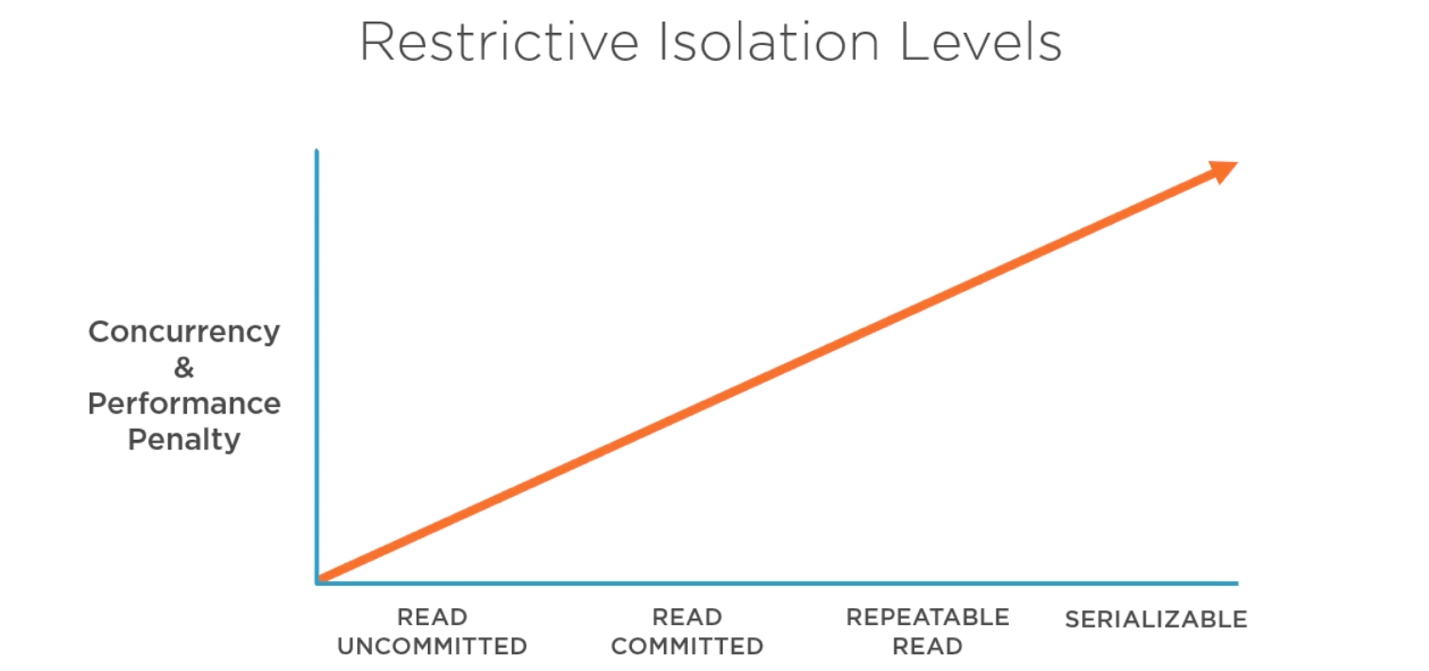
1. **Read Uncommitted**
   1. Doesn’t exist in/not supported in PostgreSQL
   2. Least restrictive isolation level
   3. All concurrency phenomena are permitted, including dirty reads, non-repeatable reads, and phantom rows
2. **Read committed**
   1. Prevents dirty reads, but non-repeatable reads and phantom rows can still occur
   2. Implemented using snapshots



1. **Repeatable Read**
   1. Prevents both dirty reads and non-repeatable reads but may still allow for phantom rows to occur
   2. In PostgreSQL, the repeatable read isolation level also prevents phantom rows



1. **Serializable**
   1. Emulating serial transaction execution as if they were executed one after another and not concurrently
   2. Expensive in terms of performance, concurrency, and also the potential for conflict to concur
   3. Prevents dirty reads, non-repeatable reads, and phantom reads



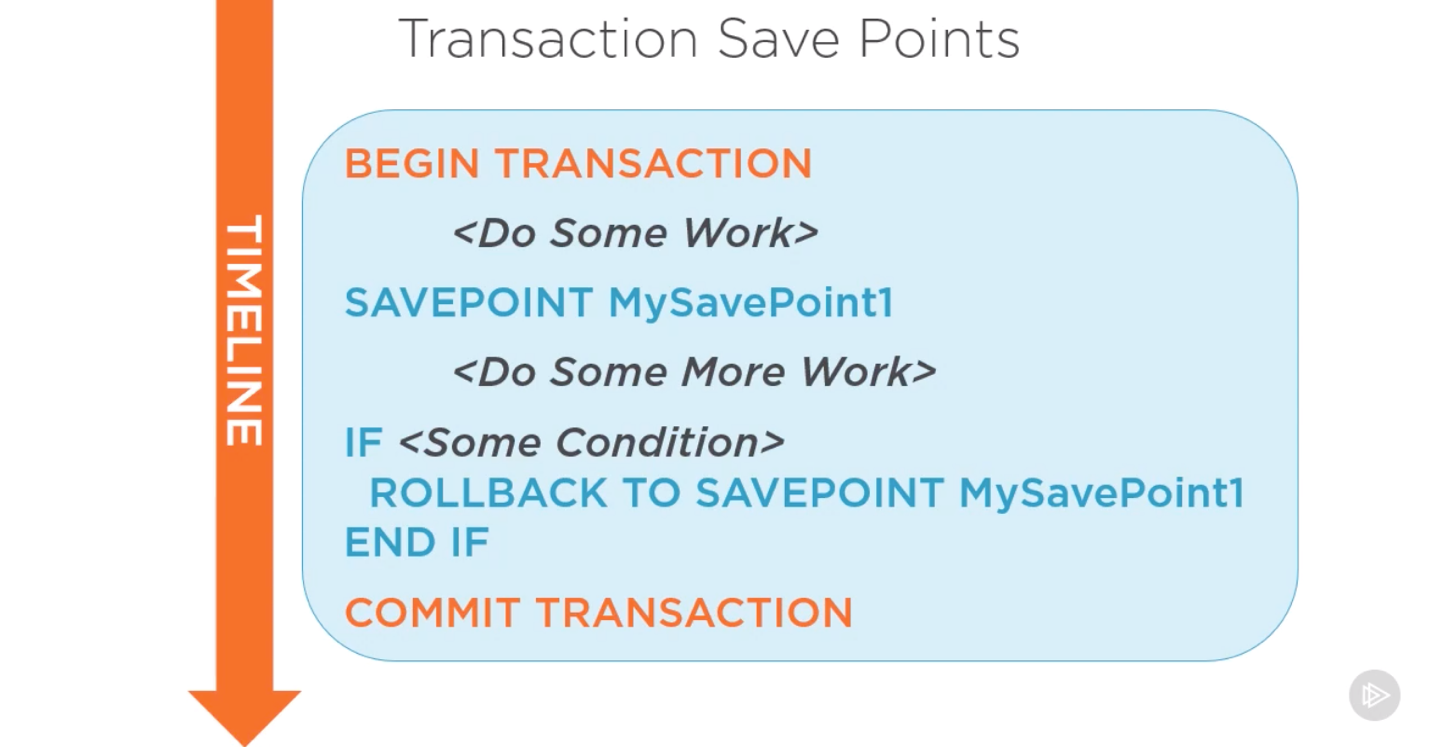
**TAKEAWAYS:**

* Isolation strategy is a crucial part of application design and will have a detrimental impact on your data consistency
* It is a complicated subject, with lots of controversy in the industry, and will require significant resources to learn and plan for
* Different areas of the application will have different isolation requirements

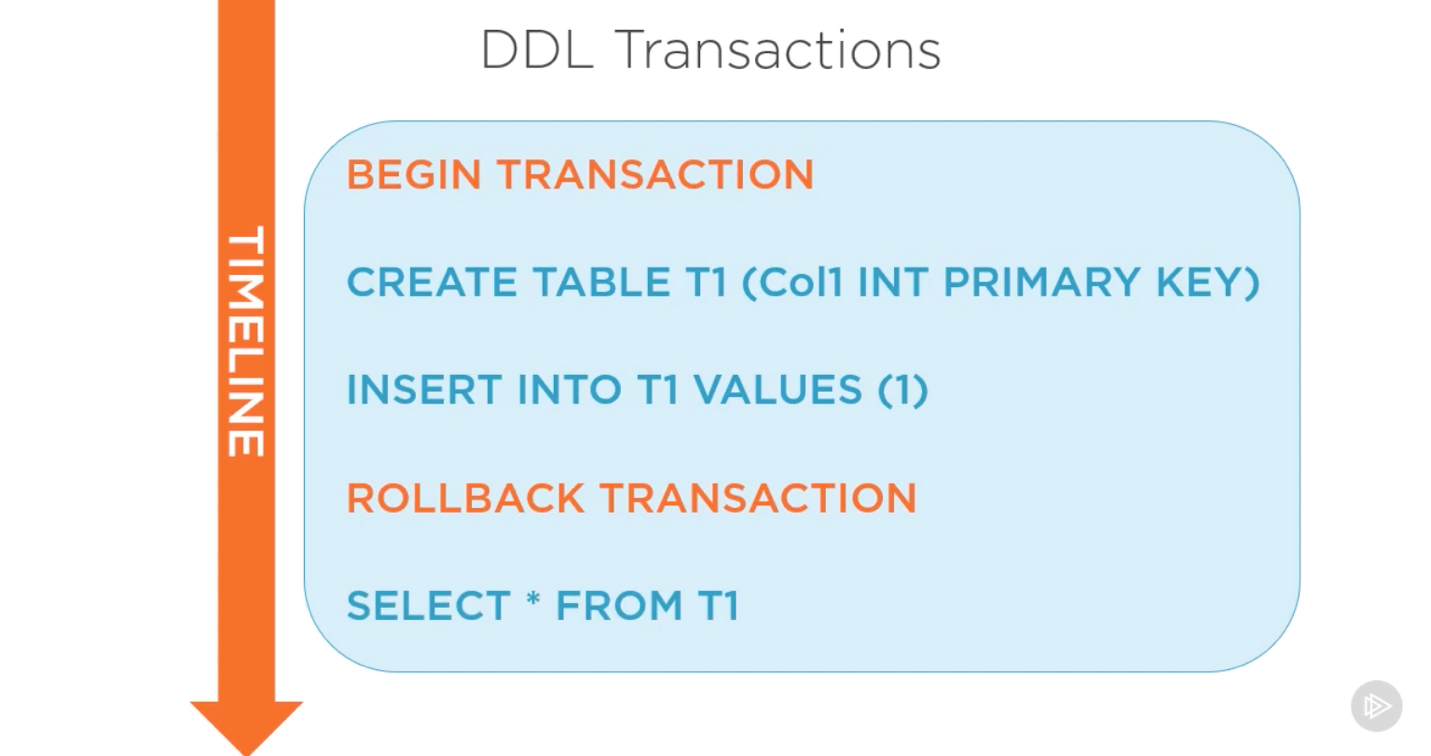
**Using Transaction Control Language**

Used to define and control our transactions properties

1. **Transaction Blocks/Boundaries**
   1. BEGIN/START to begin a transaction block
      1. BEGIN | BEGIN WORK | BEGIN TRANSACTION | START TRANSACTION
      2. BEGIN TRANSACTION ISOLATION LEVEL can be used as part of the begin statement
   2. COMMIT/ROLLBACK to terminate a transaction block
      1. COMMIT | COMMIT WORK | COMMIT TRANSACTION
      2. ROLLBACK | ROLLBACK WORK | ROLLBACK TRANSACTION
   3. Full transaction block would be:
      1. BEGIN TRANSACTION 🡪 <Transaction Body> 🡪 COMMIT TRANSACTION
      2. PostgreSQL does not support nesting of transactions; it will issue a warning
2. **SET TRANSACTION**
   1. Change isolation level
   2. Can be done inside the transaction body (i.e. setting a default) or before the BEGIN TRANSACTION statement
3. **Transaction Save Points**
   1. Save points need to be named



1. **DDL Statements**
   1. In PostgreSQL, DDL statements are fully transactional
   2. Thus, an entire DDL transaction can be rolled back
   3. In the image below, the rollback applies to both the CREATE TABLE and the INSERT statements so the SELECT statement won’t work because there will not be a T! table



**WATCH: DEMO USING TCL**

[**https://app.pluralsight.com/course-player?clipId=508464c3-b02f-4b1a-afc5-314a8370fe6e**](https://app.pluralsight.com/course-player?clipId=508464c3-b02f-4b1a-afc5-314a8370fe6e)