**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**

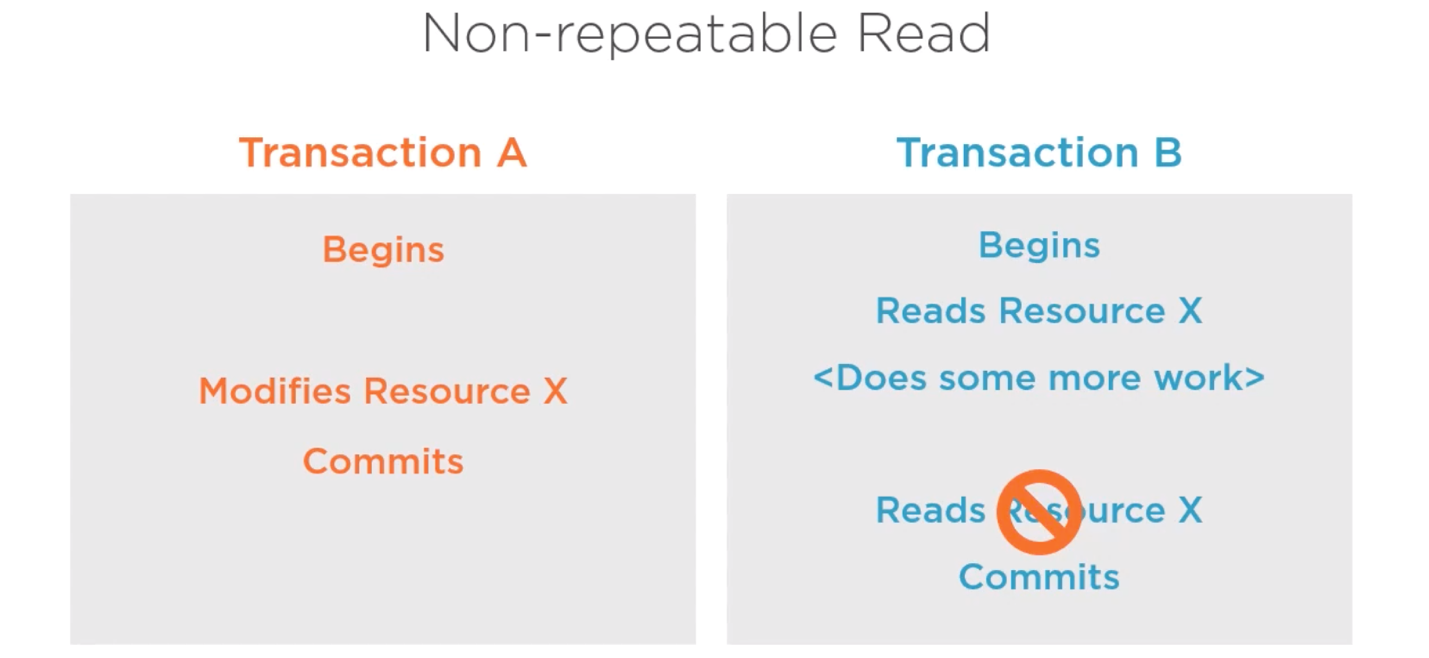
* **Pessimistic**: this approach assumes that there will be conflict so it takes measures to prevent it in advance
  + Uses locks
  + Introduces waits (blocks)
  + Uses a single copy of data
* **Optimistic**: assumes that conflicts will be rare, but requires a validity check before committing
  + Uses 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

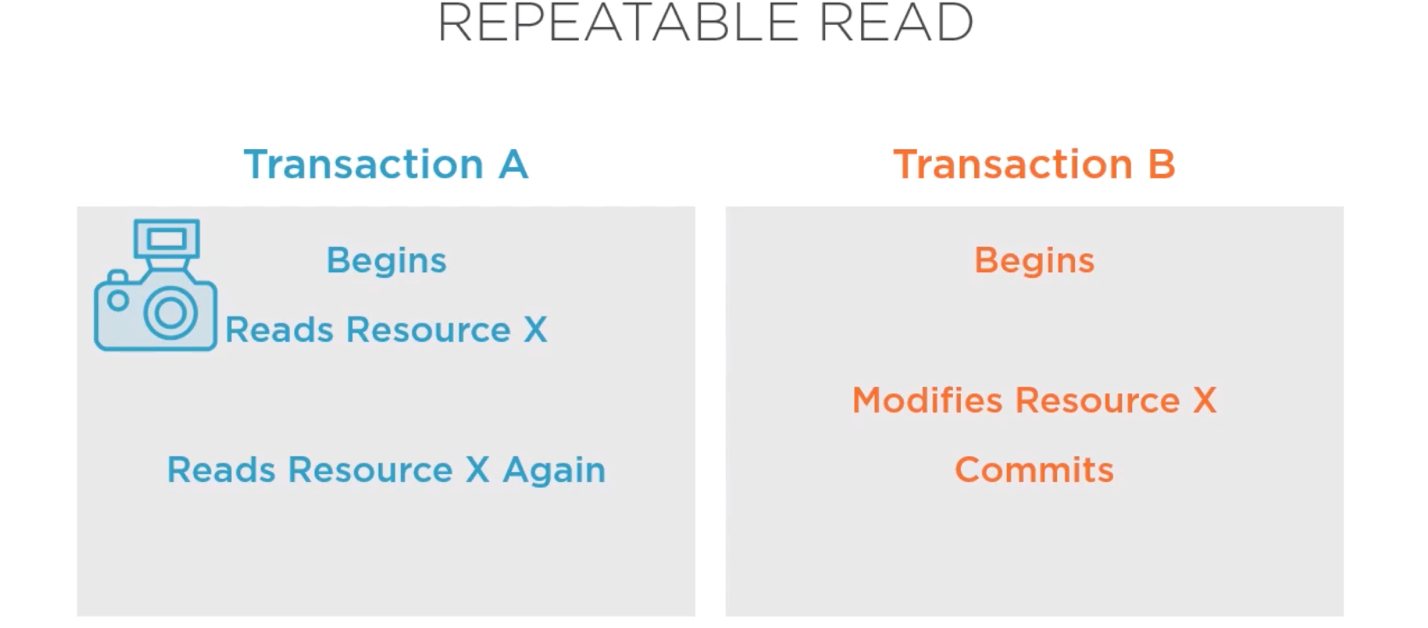


**How ANSI Isolation Levels Solve Concurrency Phenomena**

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



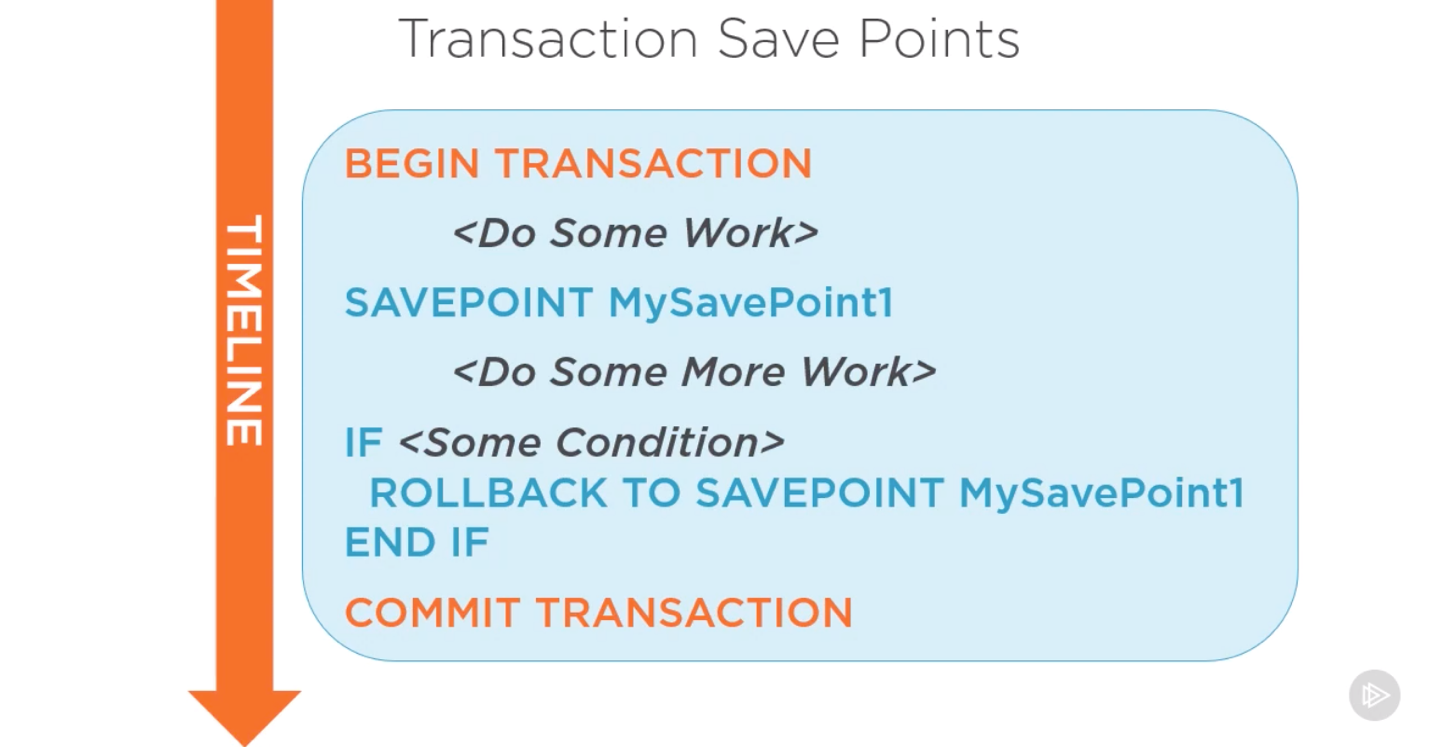
**TAKEAWAYS:**

* 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

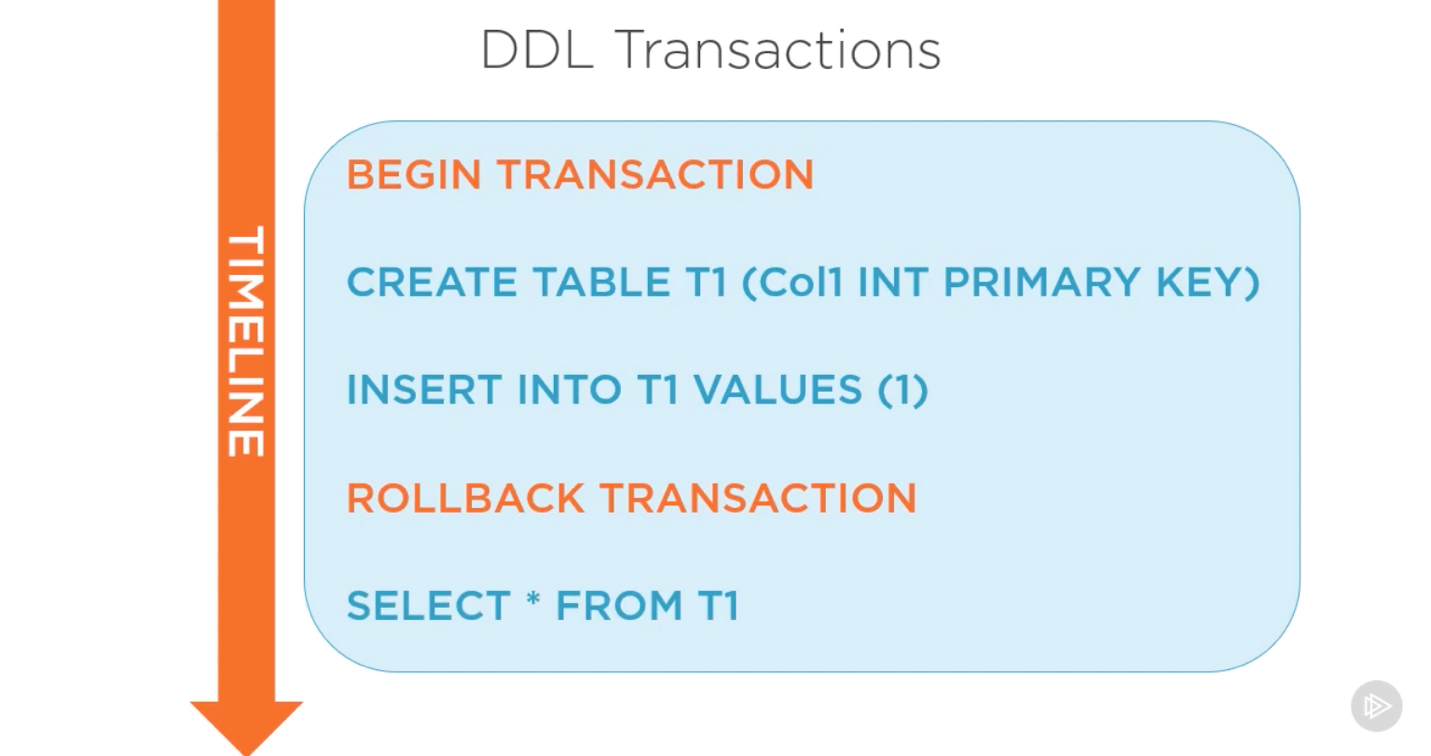
**Using Transaction Control Language**

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 help us be able to sub-divide the transaction into smaller units of work
   2. 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

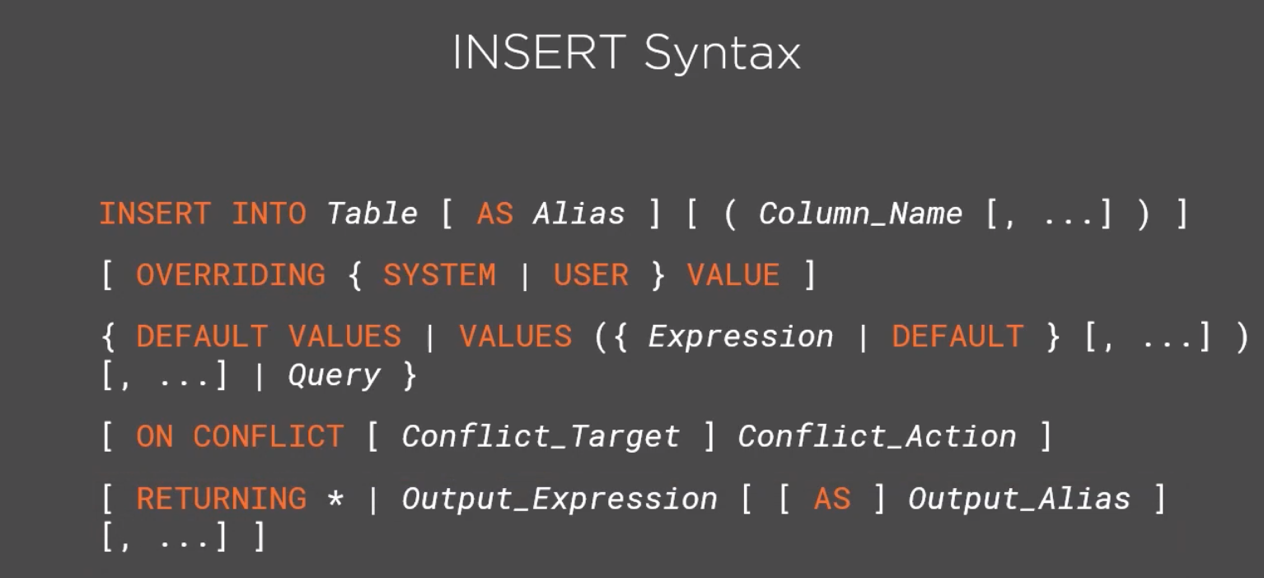


**WATCH: DEMO USING TCL**

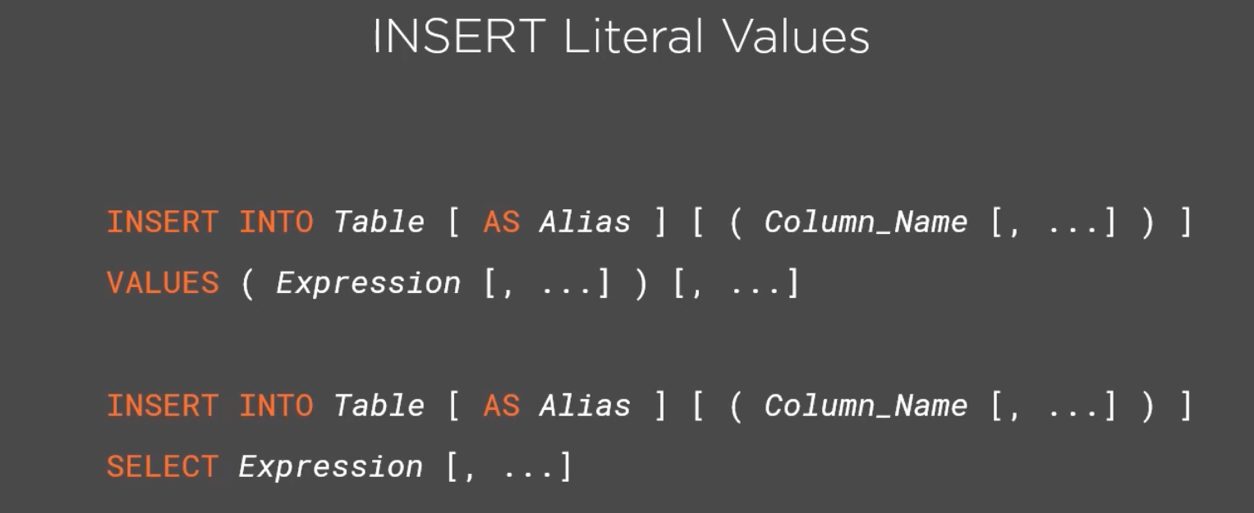
[**https://app.pluralsight.com/course-player?clipId=508464c3-b02f-4b1a-afc5-314a8370fe6e**](https://app.pluralsight.com/course-player?clipId=508464c3-b02f-4b1a-afc5-314a8370fe6e)

**Inserting Data with the INSERT Statement**

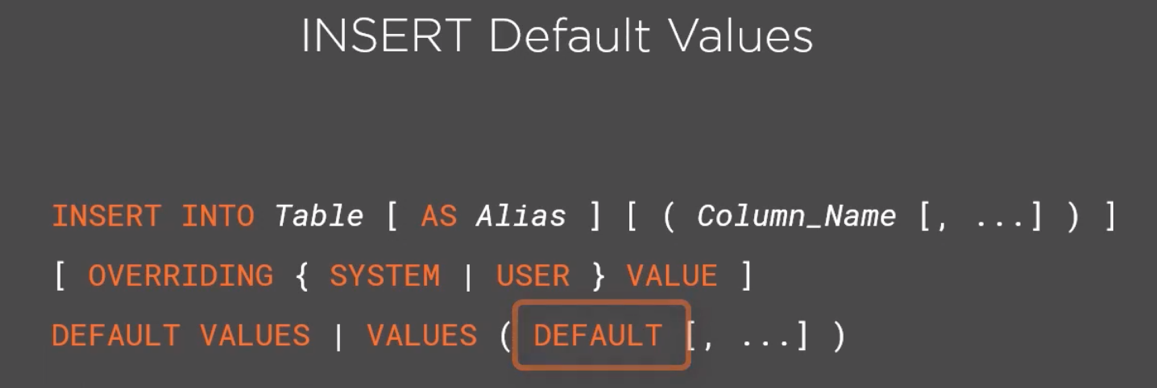
1. INSERT Syntax Overview
   1. Syntax Conventions
      1. KEYWORDS
      2. *Variables, Expressions, Aliases*
      3. [ Optional Clauses ]
      4. { Mandatory Clauses}
      5. Option 1 | Option 2
      6. Repeat = [, …]
   2. INSERT Syntax
      1. INSERT INTO (mandatory for PostgreSQL but optional for other engines)
      2. OVERRIDING clause is used to overwrite automated system values
      3. Data clauses as DEFAULT VALUE or VALUE
      4. ON CONFLICT is used to handle when our actions conflict with existing table constraints
      5. RETURNING clause allows us to not only insert the data, but return the data to the client



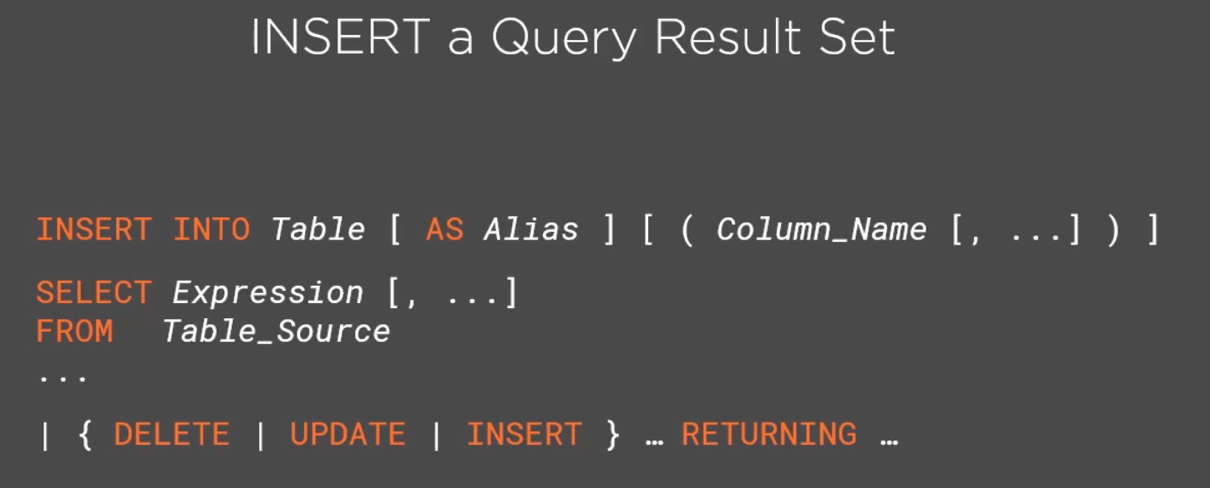
1. Inserting Literal Values
   1. The VALUES row constructor can be used to generate the data for inserted rows
   2. Can also be done using a SELECT without a FROM clause



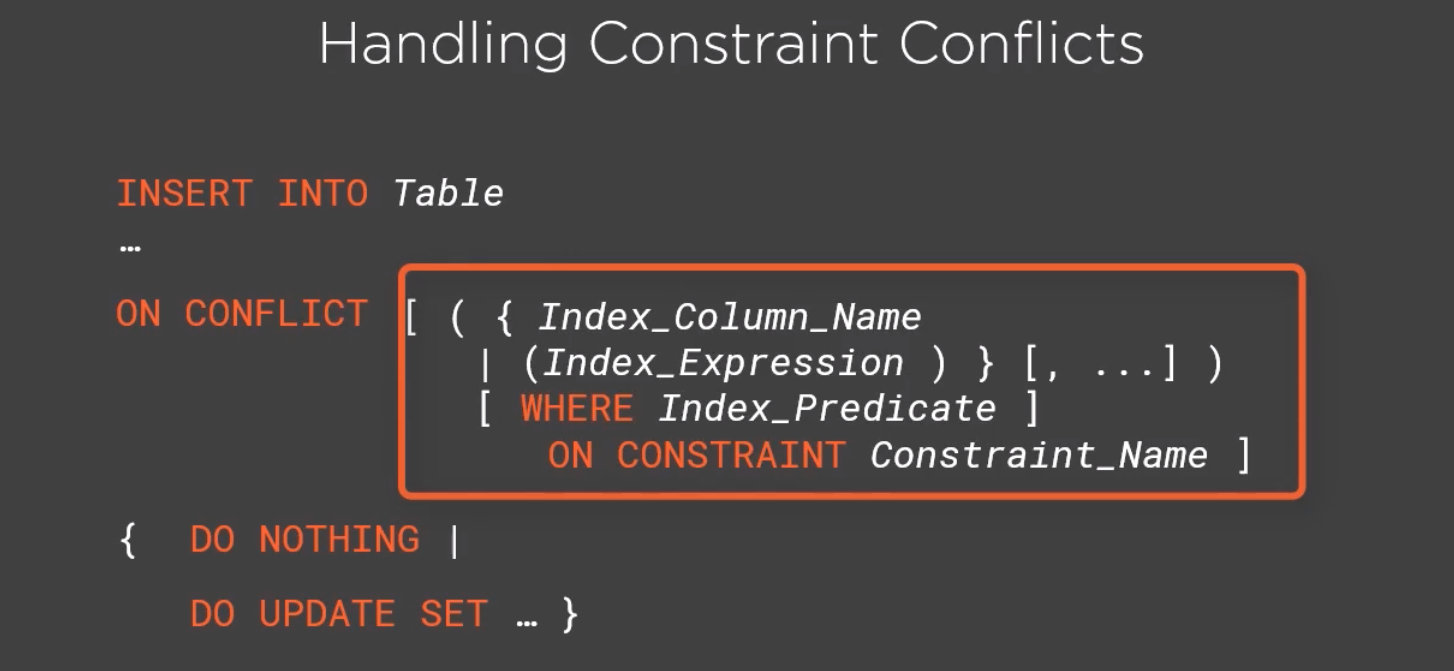
1. Inserting Default Values
   1. We do not explicit state the actual value as the insert statement; instead, we instruct the database engine to populate that value for us
   2. PostgreSQL offers two types
      1. With a default constraint that is assigned to a column
      2. An identity or sequence columns which can be used to generate sequential identifiers for rows to often be used as surrogate keys
      3. We can use them by excluding them from our INSERT statement
   3. We can use OVERRIDING when conflicts occur –either USER | SYSTEM
   4. We can also trigger default constraints is to use the DEFAULT VALUES clause when we want the whole row to be defaults, or use the keyword DEFAULT as part of the VALUES clause



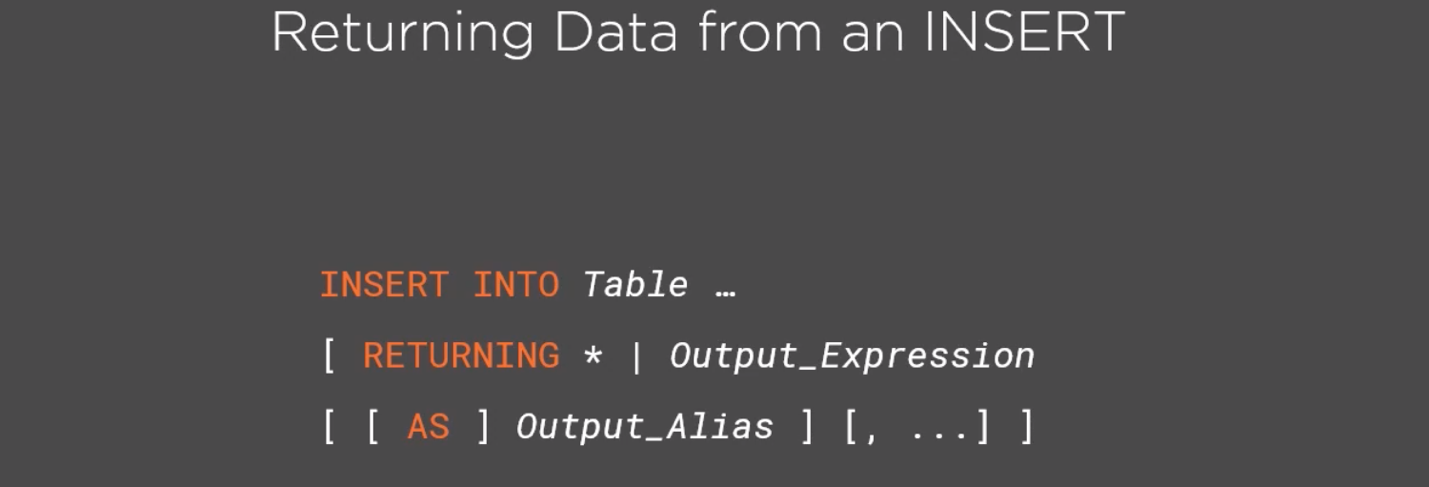
1. Inserting a Query Result Set
   1. INSERT and SELECT can be used together to create a whole set of inserts at once
   2. We take the output of a SELECT statement and use that as the input to our INSERT so we can insert whole datasets at once, including a set that consists of data from the same table that we are INSERTing into.



1. Inserting Data Through Views (and CTEs)
   1. Views are persistent code objects in the database which consist of a SELECT statement and an ALIAS (they’re schema object that consists of an aliased SELECT query)
   2. We can INSERT indirectly via views
   3. This can be used as a table source for other SQL statements
   4. They’re great for abstractions, code encapsulation, and security management
   5. Common Table Expressions are a query scoped view, defined and used only within the scope of a query
      1. PostgreSQL doesn’t allow CTEs to be used as a target of a DML
      2. PostgreSQL doesn’t allow INSERT into a CTE
   6. Updateable views (can be used as the target of a DML statement) rules
      1. SELECT query must not contain
         1. More than one table or updateable view in its FROM clause
         2. WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses
         3. UNION, INTERSECT, or EXCEPT
         4. Aggregates, window functions, or set-returning functions in the SELECT list
      2. Must be a base table or a nested updateable view, which in turn must comply with the same rules
2. Handling Constraint Conflicts
   1. Conflicts can be triggered by:
      1. PRIMARY KEY constraint (unique and NOT NULL)
      2. UNIQUE constraints
      3. EXCLUDE constraints (PostgreSQL proprietary)
   2. We can use DO NOTHING to ignore the error and let the insert continue with just the non-conflicting rows
   3. When we don’t want to ignore the error, but we want to update the existing conflicting row with the values, we need to specify a conflict action of DO UPDATE by the individual set column expressions



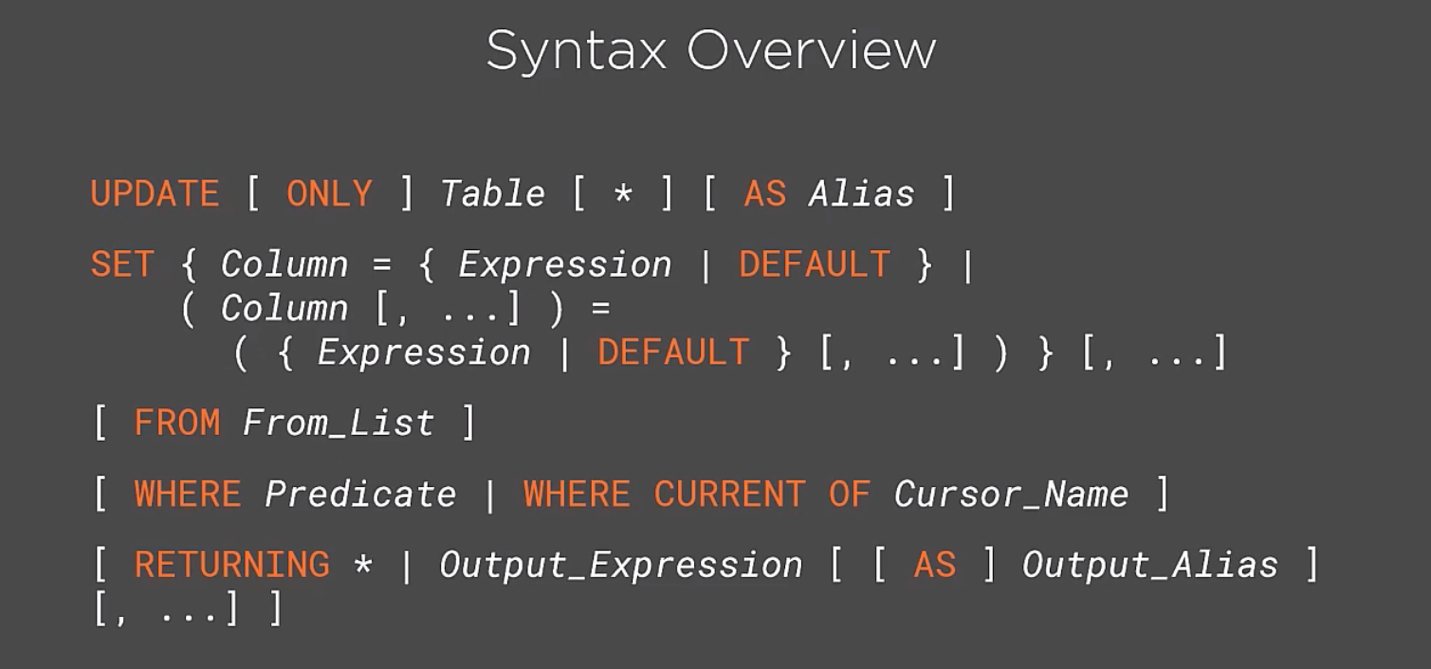
1. Returning Data from INSERT
   1. Most common purposes:
      1. Returning data from an INSERT is to retrieve auto-generated values for further processing by client application
      2. Log DML statements for auditing purposes
      3. Logging of conflict resolution



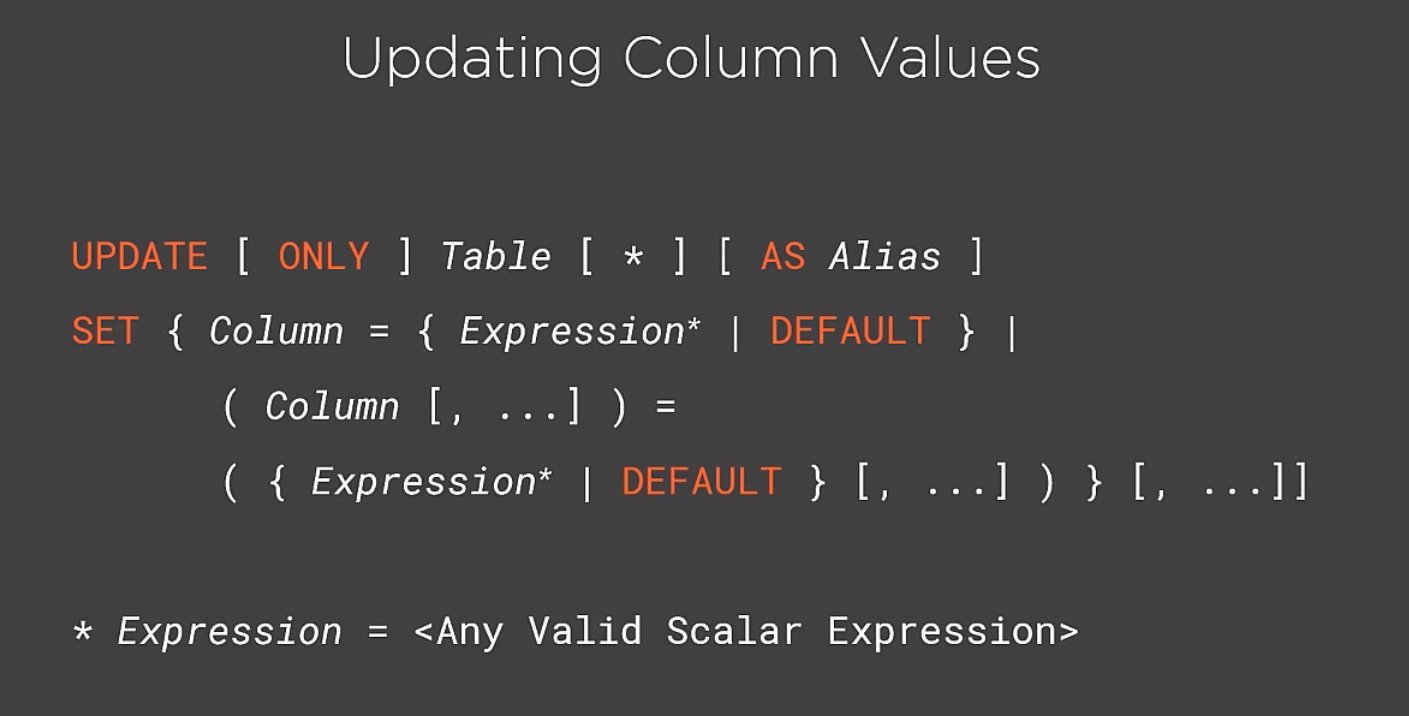
**Using the UPDATE Statement**

**Used to update existing data in tables**

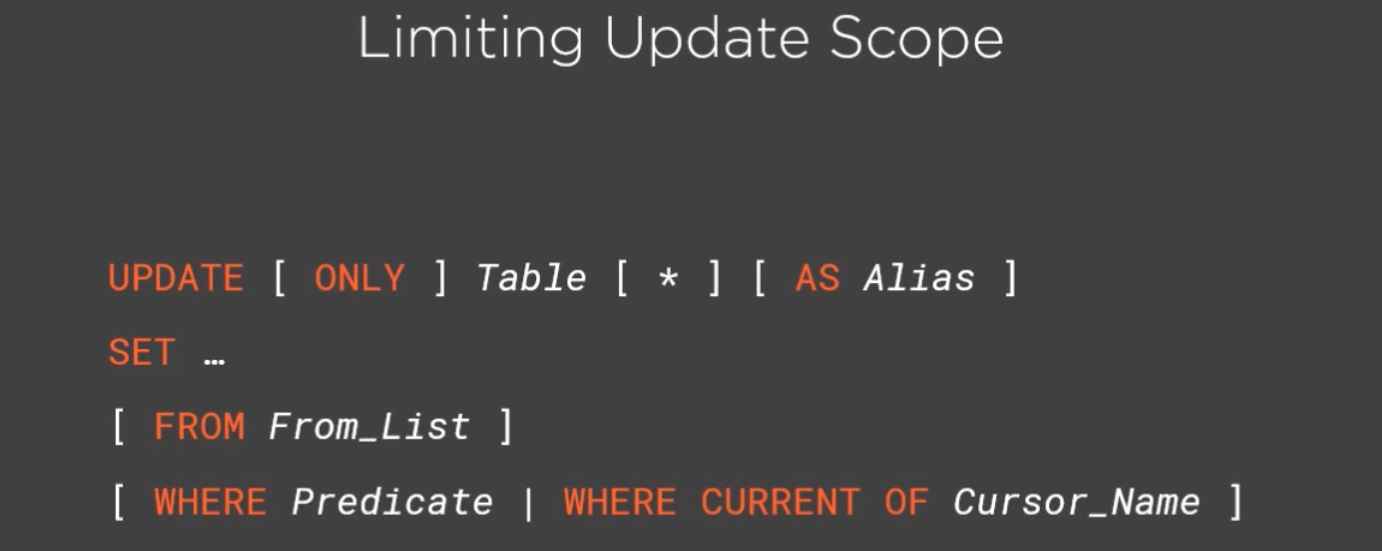
1. Syntax Overview
   1. Starts with UPDATE, followed by the optional ONLY clause
      1. AS Alias can be used to join other tables specified in the FROM clause
   2. SET clause is used to assign each column that needs to be updated
      1. Can use DEFAULT to set default values generated by the database engine
   3. FROM list let us specify additional table expressions similar to a FROM clause of a SELECT statement
   4. The WHERE clause lets us specify which rows are to be updated
   5. The RETURNING clause allows us to return data to the calling application or an outer query



1. Updating Column Values
   1. We can instruct PostgreSQL whether or not our UPDATE statement should affect tables that inherit our target table by specifying ONLY to disable target table updates or specifying the \* to explicitly state that we do want all inheriting tables to be updated
   2. The SET clause details the actual column value changes that we want to perform. The format is the Column name, the = sign for assignment, and the Expression for the new value
      1. These expressions can be any valid scalar expression, like literals, other columns from the same or other tables, or even subqueries, as long as they return only one row
      2. PostgreSQL also supports assigning multiple values for multiple columns from the same SET line by specifying both the column list and the value list in parentheses



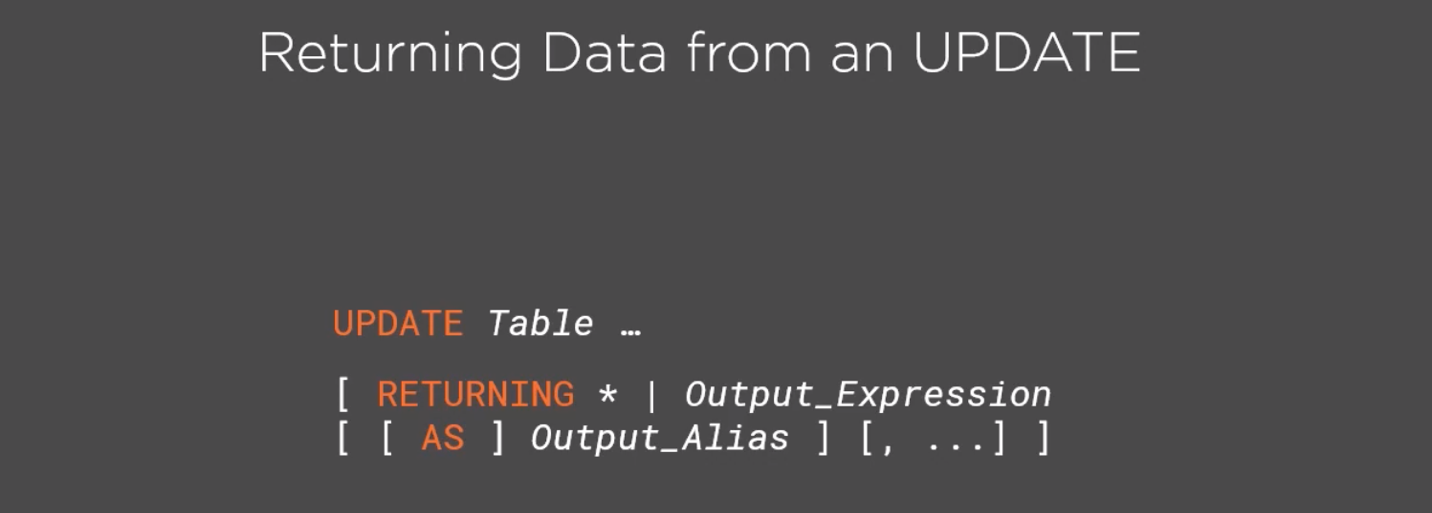
1. Limiting Update Scope Using (filters like:)
   1. FROM clause like [ FROM *From\_List* ]
   2. Predicates like [ WHERE *predicate* | WHERE CURRENT OF *Cursor\_Name* ]
   3. Cursors
   4. Other Tables



1. Updating Through Views
   1. We first create the View then use the View\_Name as the target of the UPDATE
   2. The same rules apply as for the INSERT modules

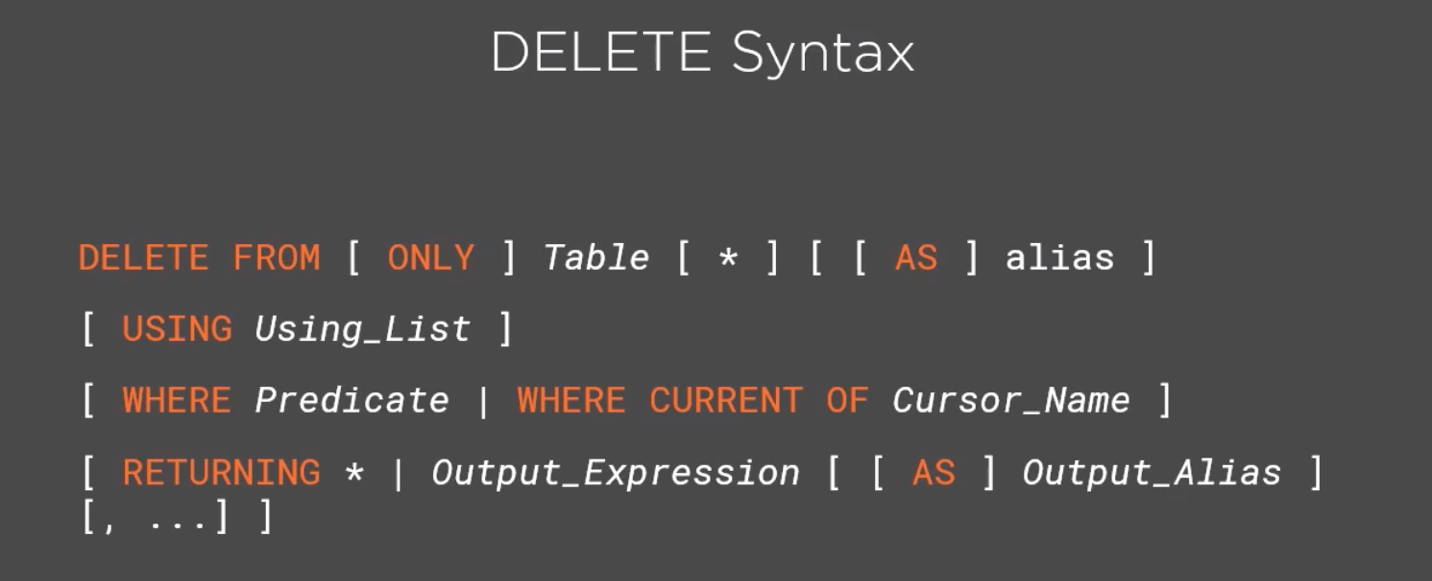


1. Returning Data from UPDATE
   1. The two main reasons to return data from an UPDATE are:
      1. Archiving updated row values as part of a data lifecycles strategy
      2. Auditing compliance requirements
   2. The syntax is the same as an INSERT; we just need the RETURNING clause at the end of the UPDATE statement and specify which expressions we’re interested in returning



**Deleting Data Using the DELETE Statement**

1. DELETE Syntax Overview
   1. Begins with a DELETE FROM keywords, followed by the optional keywords that tells the engine how to handle tables that inherit our target table
      1. We can explicitly state that we want all tables to be updated with \* sign after the target table name
      2. The ONLY keyword specifies the specific target tables to be deleted
   2. The USING clause is similar to the FROM clause in an UPDATE statement
   3. The WHERE predicate is used to provide the join condition for the table expression in the USING list and any other row filters
      1. The alternative WHERE CURRENT OF can be used to delete a row via a cursor
   4. The RETURNING clause allows us to return data to the calling application or an outer query



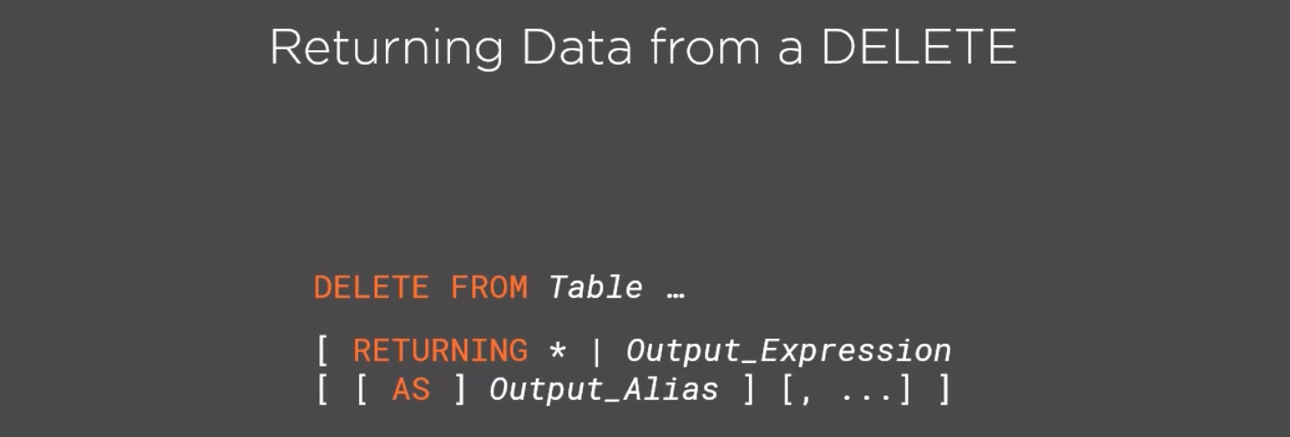
1. Limiting DELETE Scope Using
   1. Predicates
      1. Use the USING clause to virtually join our target table to another table or table expressions
   2. Cursors
      1. We should only use [ WHERE *Predicate* | WHERE CURRENT OF *Cursor\_Name* ]
      2. Deleting using cursors is not advisable
   3. Table Expressions
   4. **If we execute a DELETE statement without any filters, all rows will be deleted**

****

1. Deleting Through Views
   1. The same rules for UPDATE apply, as well as the rules for creating a View

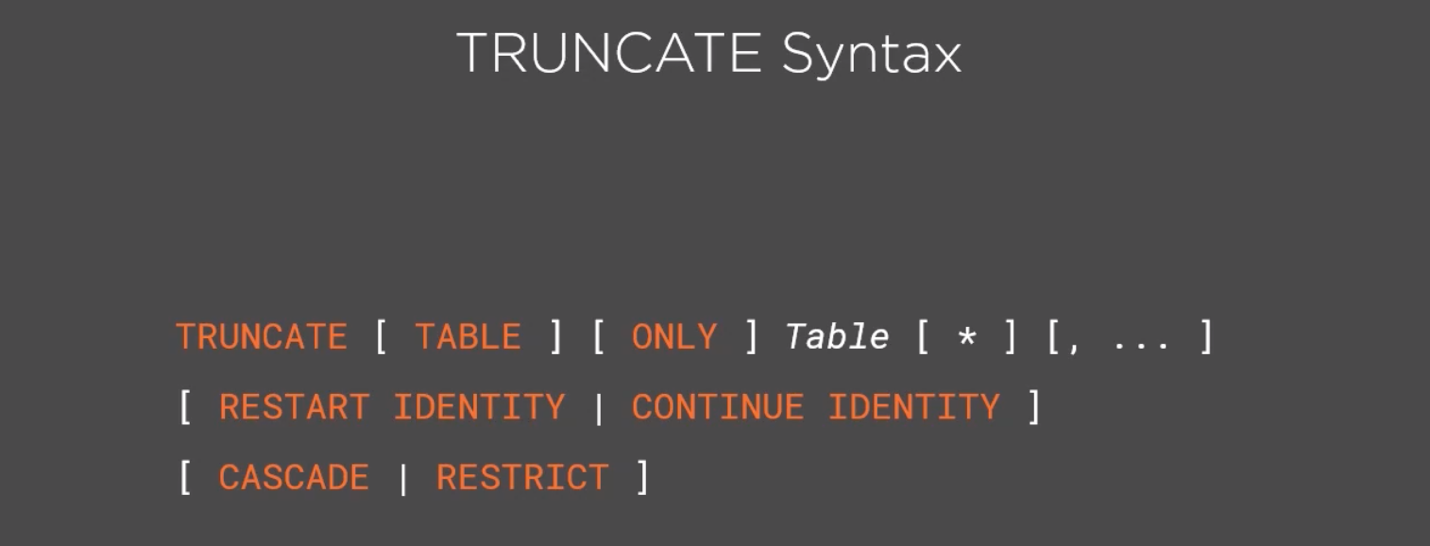


1. Returning Data from DELETE
   1. The main use cases are
      1. Archiving old data, performing the DELETE while at the same time inserting the deleted rows into an Archive table
      2. Auditing requirements that dictate that all deletes must be fully logged for compliance as well
      3. Moving data between tables—rare, like for custom partitioning

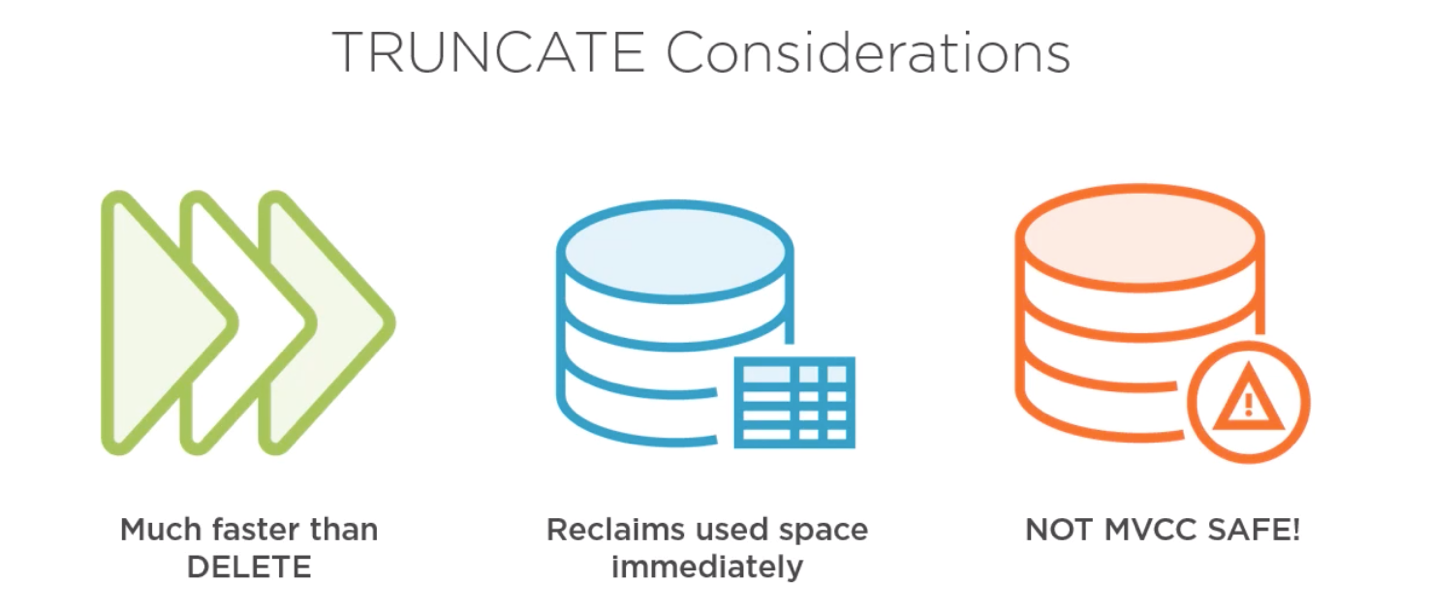


**Performing Bulk Deletes**

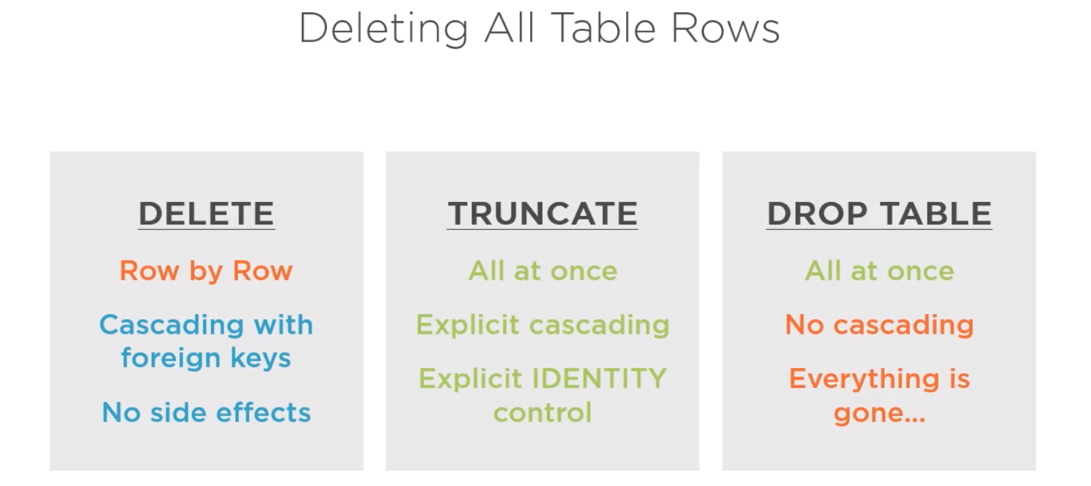
1. **The TRUNCATE Statement** 
   1. Starts with the TRUNCATE keyword followed by the Table limiting keywords
      1. [ TABLE ] [ ONLY ] *Table* [ \* ] [ , … ]
   2. Identity handling clause
      1. [ RESTART IDENTITY | CONTINUE IDENTITY ]
      2. RESTART: after the truncation, the identity properties will be reset to the seed values
      3. CONTINUE will cause the identity values to continue from their current ones
   3. We can explicitly specify that we want to truncate all related tables as well by specifying the CASCADE keyword
      1. The default is RESTRICT, which prevents this
      2. If there are foreign key dependencies, TRUNCATE without a CASCADE will fail with an error message



* 1. TRUNCATE Considerations
     1. It is much faster than a DELETE
     2. We can reclaim the used space immediately
     3. It is NOT MVCC safe



1. **Deleting** All Table Rows
   1. The key considerations when deleting all rows are:
      1. DELETE performs a row-by-row operation, potentially cascading the DELETE to child tables with foreign keys, and has no impact on identity or other properties of the table
      2. TRUNCATE improves performance significantly over the DELETE statement by deleting the entire table at once. It allows explicit control over descendant tables and child tables with foreign keys, and also allows for explicit control over identity columns
      3. The third alternative, the DDL DROP TABLE statement, which will drop the table and remove all of its data in a single operation. DROP will fail if there are any dependent objects, and it will remove the table definition and all associated permissions that were granted
      4. Need to perform a VACUUM to reclaim the space



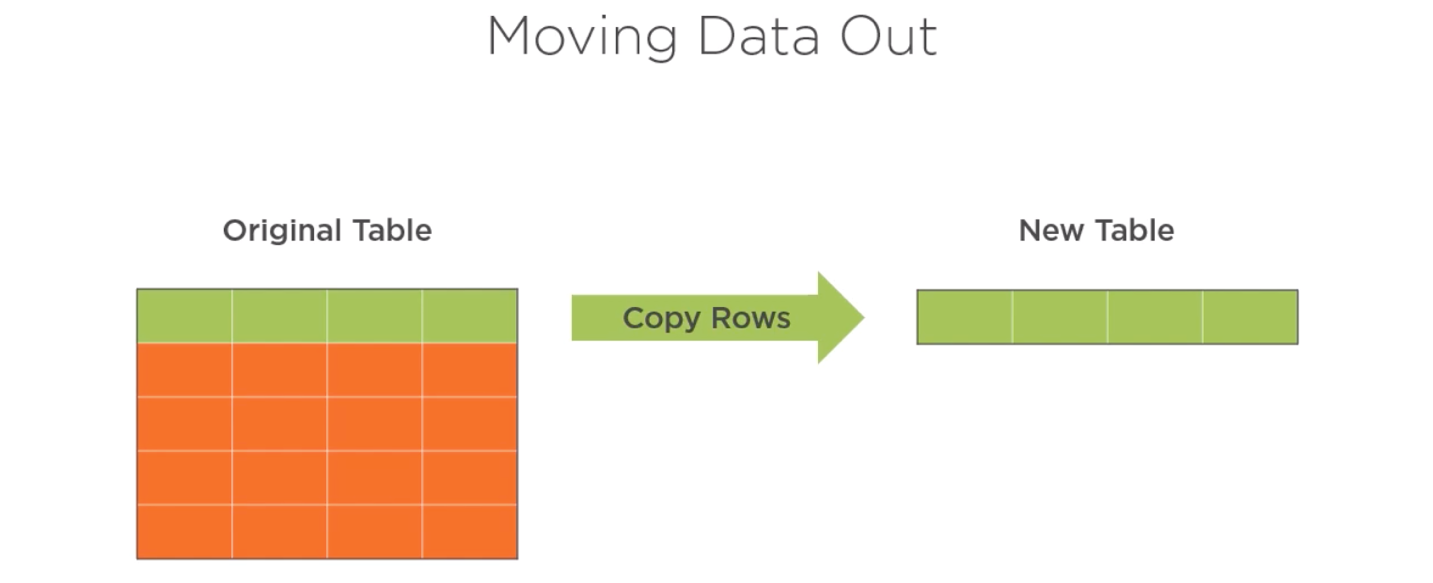
1. Large DELETE Challenges
   1. May take a very long time, hours or even days
   2. May cause excessive locking and snapshots of data to be kept for a long duration
   3. Challenging server resources which impact concurrent workload
2. Large DELETE Optimizations
   1. **Batching**
      1. If there’s no urgency in deleting the data and no potential consistency issues with partial deletes, one of the ways to mitigate the duration penalty of large deletes, is to batch the deletes in small chunks, paying the price in small installments instead of all at once



* 1. **Partitioning** 
     1. PostgreSQL also supports partitioning, which is a way to physically divide a large table into smaller partitions with very efficient methods of dealing with an entire partition at once
     2. We’ll need to create the partition tables. Once those are created, the partitioned tables can access the parent tables in PostgreSQL
        1. Bounding partitions can be created using the MINVALUE and MAXVALUE keywords that PostgreSQL provides



* 1. **Moving Out Data**
     1. If the numbers of rows to keep is small, we can move the data we want to keep out of the table, and either TRUNCATE or DROP to eliminate the whole table, and use the copy of the rows we exported to populate the table again



* + 1. After removing the table, we can DROP the original table and rename the new table to the original table name. In this case, we would have to detach and reestablish all relationships, constraints, permissions, and supporting objects to complete the process
    2. A better alternative is to TRUNCATE the original table, leaving it empty, and copy back the rows that we wanted to keep from their temporary safe haven. This way, all the relationships, constraints, permissions, and supporting objects need not be reestablished

**Course Takeaways**

Concurrency and isolation are serious and complex challenges for RDBMS.

Getting it wrong may impact data consistency. Getting isolation wrong can inflict the ultimate damage to any organization, potentially corrupting the most valuable assets, which is the data, which cannot be replaced.

Modifying data can be expensive, but is manageable.