**Views**

A view can include all or some of the columns from one or more base tables or existing views. Creating a view creates a named specification of a results table, which can be queried in the same way as a table. You can also change the data in the base table by running insert, update, and delete queries against the view. When you define a view, the definition of the view is stored. The data that the view represents is stored in the base tables, not by the view itself. You can use a view to: Show a selection of data for a given table, so you can omit sensitive data like tax information, birth dates, or salaries. Combine two or more tables in meaningful ways. Simplify access to data by granting access to a view without granting access to the underlying tables. Show only the portions of data relevant to the process that uses the view. For example, you can create a view that displays only non-sensitive data from the Employees table; Employee ID, name, address, job ID, manager ID, and department ID. The view does not show sensitive data like salary or birthdate. You use the CREATE VIEW statement to create a view based on one or more tables or views. To define a view, use the CREATE VIEW statement and assign a name (up to 128 characters in length) to the view. List the columns that you want to include. You can use an alias to name the columns if you wish. Use the AS SELECT clause to specify the columns in the view, and the FROM clause to specify the base table name. You can also add an optional WHERE clause to refine the rows in the view. This CREATE VIEW statement <click> creates a view called EMPINFO based on the Employees table. The SELECT statement returns the data in the view, as shown in the table below. Views are dynamic; they consist of the data that would be returned from the SELECT statement used to create them. When you use a view in another SQL statement, it behaves as though you have used a SELECT statement that returns the content of the view. The SELECT statement that you use to create the view can name other views and tables, and it can use the WHERE, GROUP BY, and HAVING clauses. It cannot use the ORDER BY clause or name a host variable. In this example the EMPINFO view is created with only the rows where the MANAGER\_ID is 30002. You can use a SELECT statement to show the information from the view, and verify that only rows where the MANAGER\_ID is 30002 are included. If you need to remove a view completely, <click> you can use DROP VIEW. In this video, you learned that: Views are an alternate way of accessing data in tables. They can include specified columns from multiple base tables and existing views. Once created, views can be queried like a table, and the data in the base table can be modified through the view. Views are dynamic; only the definition of the view is stored, not the data. You can use the CREATE VIEW statement to create a view based on one or more tables or existing views.

**Stored Procedures**

A stored procedure is a set of SQL statements that are stored and executed on the database server. So instead of sending multiple SQL statements from the client to server, you encapsulate them in a stored procedure on the server and send one statement from the client to execute them. You can write stored procedures in many different languages. For example, for Db2 on Cloud and DB2 you can write in SQL PL, PL/SQL, Java, C, or other languages. They can accept information as parameters, perform create, read, update, and delete (CRUD) operations, and return results to the client application. The benefits of stored procedures include: Reduction in network traffic because only one call is needed to execute multiple statements. Improvement in performance because the processing happens on the server where the data is stored, with just the final result being passed back to the client. Reuse of code because multiple applications can use the same stored procedure for the same job. Increase in security because a) you do not need to expose all of your table and column information to client-side developers and b) you can use server-side logic to validate data before accepting it into the system. Remember though, that SQL is not a fully-fledged programming language, so you should not try to write all of your business logic in your stored procedures. So let’s look how to create a stored procedure on Db2 on Cloud in SQL. Firstly, you use the CREATE PROCEDURE statement, specifying the name of the procedure and any parameters which it will take. In this example, the UPDATE\_SAL procedure will take an employee number and a rating which it will use to update an employee’s salary by an amount depending on their rating. Then you declare the language you are using. You then enclose your procedural logic in the BEGIN END statements. In this case, giving employees who have a rating of 1 a 10% pay rise and all others, a 5% pay rise. Notice that you can use the information passed to the procedure, the parameters, directly in your procedural logic. You can call stored procedures from your external applications or from dynamic SQL statements. To call the UPDATE\_SAL stored procedure that we just created, you use the CALL statement with the name of the stored procedure and pass the required parameters – in this case, the employee id and the rating for that employee. In this video, you learned that: Stored procedures are a set of SQL statements that execute on the server Stored procedures offer many benefits over sending SQL statements to the server You can use stored procedures in dynamic SQL statements and external applications.

**ACID Transactions**

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 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 transaction in the example are called ACID transactions. 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. 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. 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. In this video, you learned that: A transaction represents a complete unit of work, which can be one or more SQL statements. An ACID transaction is one where all the SQL statements must complete successfully or none at all. This ensures the database is always in a consistent state. ACID stands for Atomic, Consistent, Isolated, Durable. SQL commands BEGIN, COMMIT, and ROLLBACK are used to manage ACID transactions. SQL commands can be called from languages like C, R and Python.

**Join Overview**

A simple Select statement retrieves data from one or more columns from a single table. The next level of complexity is retrieving data from two or more tables. This leads to multiple possibilities of how the result set can be generated. To combine data from two tables, you use the JOIN operator. A JOIN combines the rows from two or more tables based on a relationship between certain columns in these tables. In this simplified library database example, author and book are entities. This entity relationship diagram represents the relational data model for the author and book entity as well as other entities such as borrower, loan, copy, and author list. The information is split into different tables. If you wanted to know which borrower has which copy of a book out on loan, you need to gather data from three tables: the borrower, loan, and copy tables. This is when you need to use the JOIN operator. First you need to identify the relationship between these tables. That is, the column or columns in each table to use as a link between the tables. In this entity relationship diagram, notice the author ID, book ID, borrower ID, and copy ID have the primary key icon. A primary key uniquely identifies each row in a table. Notice also the entities on the lower half of the screen some attributes have FK in brackets next to them. This identifies a foreign key, which is a set of columns referring to a primary key of another entity. For example, the loan entity has the borrower ID attribute with the FK in brackets. In this example, the borrower ID attribute is the Foreign Key in the loan entity, which refers to the Primary Key for the borrower entity. So, if you wanted to know which borrower has a book out on loan, you need to gather data from the borrower and loan tables.You will need the borrower ID from both tables. So far, you have seen an example of combining two tables. But what if you need to combine data from three or more different tables? You simply add new tables to the joins. For example, if you want to know which borrowers have a book on loan, and which copy of the book they have on loan, first you join the information from the borrower table and the loan table by matching borrower IDs. Then, you join the information from the loan table and the copy table by matching the copy IDs. SQL offers you several different types of JOINs. You can extract a data set corresponding to the intersection of the two tables involved, or you can choose a bigger data set. You can go up to the point of selecting the combination of all the data from these two tables. The most common type of join is an inner join, which displays only the rows from two tables that have matching value in a common column, usually the primary key of one table that exists as a foreign key in the second table. There are also outer joins, which return matching rows, and even the rows from one or the other table that don’t match. There are many varieties of outer join that you can use to refine your result set. In this video, you learned that: You can use the JOIN operator to combine rows from two or more tables The tables being joined are related by a common column, which is usually the primary key of one table, and appears as a foreign key in the other table There are two types of joins; inner joins and outer joins.

**Inner Join**

A join operation combines the rows from two or more tables based on a relationship between certain columns in these tables. There are two types of table joins: inner joins and outer joins. The most common type of join is an inner join, which displays only the rows from two tables that have matching value in a common column, usually the primary key of one table that exists as a foreign key in the second table. This is the syntax of the select statement for an inner join. Imagine you want to retrieve a list of all people who are borrowing books, and the date of the loan. You need data from the borrower table and the loan table. In the FROM clause, you specify the join between the borrower table and the loan table as BORROWER INNER JOIN LOAN. You identify the borrower table as B, and the loan table as L. The table specified on the left of the JOIN clause is known as the left table – in this case, the borrower table is the left table. For this join, you select borrower ID, last name, and country from the borrower table, and the borrower ID and the loan date from the loan table. In the ON clause, you specify the JOIN predicate, in this case the condition that the borrower ID in the borrower table is equal to the borrower ID in the loan table. Notice that in this join each column name is prefixed with either the letter B or L. In SQL, this is referred to as an alias. Using an alias is much easier than rewriting the whole table name. The result set shows only the rows from both tables that have the same borrower ID. The rows are displayed if they Borrower\_Id matches. Rows with Borrower\_IDs that do not match are not displayed. The Borrower\_Id, Lastname, and Country columns are taken from the Borrower table and joined to the Borrower\_Id and Loan\_Date columns from the Loan table to make the result set. In this video, you learned that: Inner joins return only the rows from the tables that have matching value in a common column, usually the primary key of one table that exists as a foreign key in the second table. Rows from joined tables that do not have a matching value do not appear in the result.

**Outer Join**

Unlike inner joins, outer joins also return the rows that do not have a match between the tables. SQL offers you three types of outer joins: left outer join, right outer join and full outer join. In a left outer join, all the rows from the first table (on the left side of the join predicate) are included, and only the matching rows from the second table (on the right side of the join predicate). In this diagram, a Left Join matches all the rows from the left table and combines the information with rows from the right table that match the criteria specified in the query. In a right outer join, all the rows from the first table (on the left side of the join predicate) are included, and only the matching rows from the second table (on the right side of the join predicate). In this diagram, a Right Join matches all the rows from the right table and combines the information with rows from the left table that match the criteria specified in the query. A full join returns all rows from both the right table and the left table. So, the FULL JOIN can return a very large result set. In this diagram, the result set of a RIGHT JOIN is all rows from both tables matching the criteria specified in the query, plus all non-matching rows from the RIGHT table. This is the syntax of the SELECT statement for a LEFT JOIN. In this example, the Borrower table is the first table specified in the FROM clause of the SELECT statement, so the Borrower table is the LEFT table, and the Loan table is the RIGHT table.

In the FROM clause, Borrower is listed on the left side of the join operator, therefore you will select all rows from the Borrower table and combine them with the contents of the Loan table based on the criteria specified in the query. In this example, the criteria is the BORROWER ID column. For a LEFT OUTER JOIN, simply called a LEFT JOIN, you will select the following columns from the Borrower table: BorrowerID, LastName, and Country, and you will also select the following columns from the Loan table: BorrowerID, and LoanDate. The LEFT JOIN selects each BORROWER ID in the Borrower table and displays the LoanDate from the Loan table. The result set shows each Borrower ID from the borrower table, and the loan date for that borrower. There is no loan date for the last three rows, so the borrower ID and loan date show null values. This is the syntax of the SELECT statement for a RIGHT JOIN. In this example, the Borrower table is the first table specified in the FROM clause of the SELECT statement, so the Borrower table is the LEFT table, and the Loan table is the RIGHT table.

In the FROM clause, the Loan table is listed on the right side of the join operator, therefore you will select all rows from the Loan table and combine them with the contents of the Borrower table based on the criteria specified in the query. In this example, the criteria is the BORROWER\_ID column. For a RIGHT JOIN, you will select the following columns from the Loan table: Borrower\_ID, and LoanDate, and you will also select the following columns from the Borrower table: Borrower\_ID, LastName, and Country where the Borrower\_ID in the Loan table matches the Borrower\_ID in the Borrower table. The result set shows each Borrower ID from the Loan table and the Loan Date for that Borrower, where the Borrower ID in the Loan table also exists in the Borrower table. For the last row, there is no matching row in the borrower table, so the Borrower\_ID, Lastname, and Country show null values. This could indicate a problem for the library; it indicates there is a book on loan to an unknown person. This is the syntax of the SELECT statement for a FULL JOIN. For a FULL JOIN, you select all rows from the Borrower table and all rows from the Loan table. The result set shows all eight records from the Borrower table listed with the corresponding data from the Loan table. Once again, three rows return a NULL value because Borrowers Peters, Li, and Wong have never taken a book out on loan. The last row returns values for Borrower\_ID and Loan\_Date from the Laon table, but returns NULLs from the Borrower table. In this instance, there is no match in the Borrower table – the borrower of this book is unknown. In this video, you learned that: There are many varieties of outer join that you can use to refine your result set. Left outer joins return all rows from the left table, and all the rows form the right table that match that an inner join would return and all the rows in the first table that do not have a match in the second table. Right outer joins return all the rows that an inner join would return and all the rows in the second table that do not have a match in the first table. Full outer joins return all matching rows from both tables and all the rows from both tables that don’t have a match.