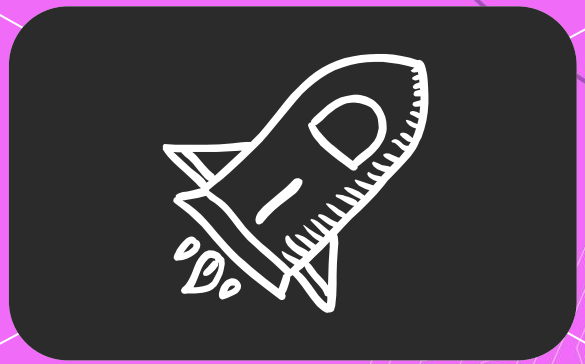
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Semana 8
¡Bienvenidos!



Temas de hoy

1 MER a Tablas

2 SQL Básico

- SELECT
- WHERE, ORDER BY
- JOINS

3 SQL Avanzado

- Subconsultas
- SUM, COUNT, AVG, etc.
- Transacciones y control de concurrencia

1

MER a Tablas

Aprendimos a diagramar nuestro diseño conceptual denotando las entidades y atributos principales de nuestro sistema y cómo se relacionan entre sí utilizando el diagrama entidad-relación, aunque esto no era más que la representación de alto nivel de nuestra futura base de datos.

Ahora vamos por la siguiente etapa en el proceso de diseño: **El diseño lógico**. 👉

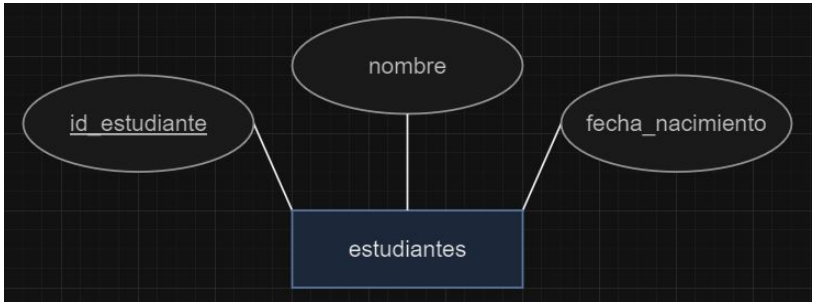
MER a Tablas

¿Qué es el diseño lógico?

El **diseño lógico** toma el Diagrama Entidad-Relación del diseño conceptual y lo transforma en un modelo más **detallado y formal** para poder implementarlo en un sistema de gestión de bases de datos relacionales **(RDBMS)**.
El proceso de convertir un MER a tablas de una base de datos implica identificar las entidades, atributos y relaciones, y luego estructurarlas en tablas siguiendo estas **reglas:**

- Cada **entidad** se convierte en una tabla.
- Los **atributos** de la entidad pasan a ser las columnas de la tabla.
- Las **relaciones** entre entidades tienen diferentes comportamientos dependiendo de su **cardinalidad**.

Veamos con los siguientes ejemplos 💥



estudiantes	
PK	id_estudiante
	nombre
	fecha_nacimiento

La entidad *estudiantes* pasa a ser el nombre de la tabla y sus atributos, sus respectivas columnas.
A las columnas se las **enlista** de forma consecutivas una debajo de la otra.
Y el **ID** que definimos con la Notación de Chen lo posicionamos al comienzo, seguido de las siglas **PK** (Primary Key)

MER a Tablas: Relaciones entre tablas

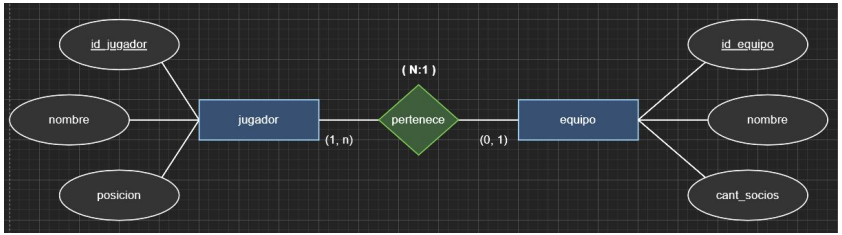
Relación Uno a muchos (1:N)

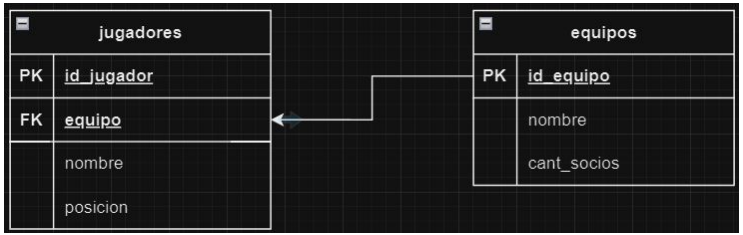
Las **relaciones** entre entidades se reflejan más detalladamente en las tablas y la cardinalidad analizada en el MER toma un papel fundamental.

Si la relación es de **uno a muchos** ó de **muchos a uno**, la clave primaria **(PK)** de la tabla del lado **"uno"** se añade como clave foránea **(FK)** en la tabla del lado **"muchos".**

Como podemos observar en el ejemplo, un jugador puede pertenecer a un equipo o a ninguno **(0, 1)** y un equipo puede tener tanto un jugador como muchos **(1, n)**, dándonos como resultado una cardinalidad de muchos a uno **(N:1)**.

Por lo tanto, la **clave primaria** de la tabla de donde proviene el **"1"** de la cardinalidad resultante *(equipos)* se agrega como una nueva columna en la tabla de **"N"** *(jugadores)* como una **clave foránea.**



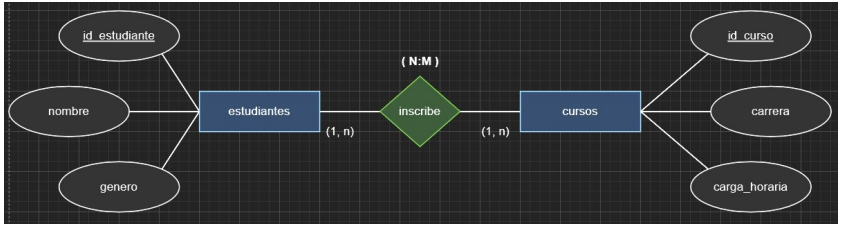


MER a Tablas: Relaciones entre tablas

Relación muchos a muchos (M:N)

Cuando hablamos de relaciones de muchos a muchos **(M:N)** existe un **conflicto** en el guardado de los datos. Esto se debe a que sin importar la tabla que elijamos, **duplicaremos** los datos y las consultas y el mantenimiento de nuestra base de datos se **dificultaría** por la redundancia de datos.
Es por ello, que para este tipo de cardinalidad debemos crear una nueva tabla **(tabla intermedia)** que contenga las **claves primarias** de las **dos** tablas relacionadas.

En este ejemplo, un estudiante puede estar inscripto en uno o en muchos cursos **(1, n)**, y a su vez, un curso puede tener tanto uno como muchos estudiantes **(1, n)**.
Por lo tanto, debido a su relación **n:m** debemos crear una tabla intermedia llamada **inscripciones** que guardará las claves primarias de ambas tablas y la fecha en que se inscriben.
Pensando un poco más a futuro, cuando queramos ver los estudiantes inscriptos, vamos a poder consultar a esta nueva tabla por dicha información, y las tablas de estudiantes y cursos permanecen con sus datos íntegros.



Entonces...

... habiendo hecho un análisis de requisitos completo y un diagrama entidad-relación **preciso** y **bien diseñado** teniendo en cuenta todos los conceptos aprendidos, el pasaje a tablas nos servirá para **visualizar** y **organizar** más detalladamente las **relaciones** y **estructura** de nuestra base de datos dejándonos un resultado óptimo para ser implementado en nuestro **RDBMS**.

2

SQL Básico

A través de SQL vamos a poder **acceder** a los datos de nuestra base de datos relacional, **filtrarlos**, **ordenarlos** y/o **combinarlos** desde diferentes tablas a nuestro antojo para tener la información de forma más **precisa** y **completa**.

Veamos cómo utilizar este nuevo lenguaje 😌

Sentencia SELECT

Antes de arrancar con las siguientes consultas, te vamos a dejar en video, la instalación de una herramienta que nos va a permitir realizar consultas SQL de forma muy visual (Workbench - Vas a poder encontrar el video en campus).

Ahora nos enfocaremos en los aspectos fundamentales de SQL que son cruciales para el trabajo diario con base de datos, y para ello, trabajaremos con una ya creada con datos almacenados para una mejor demostración.

En nuestra base de datos tenemos las siguientes tablas:

- ➔ estudiantes
- ➔ cursos
- ➔ profesores
- ➔ inscripciones

Imaginemos que es nuestro primer día de trabajo en la universidad y queremos ver **qué** información guarda la tabla **estudiantes**.
Para poder consultar y obtener información de una tabla en particular utilizaremos la sentencia **SELECT.**

```
SELECT * FROM estudiantes;
```

La sentencia **SELECT** seguida por un asterisco **"*"** lo utilizamos para seleccionar **todos** los campos (columnas) de una tabla.
Luego con la cláusula **FROM** indicamos de **qué** tabla deseamos obtener esa información, en nuestro caso, la tabla ***estudiantes***.

¡Importantísimo! Todas las líneas de ejecución en SQL deben terminar con punto y coma **";"**

Dándonos como resultado:

id_estudiante	nombre	email	fecha_nacimiento
1	María López	maria.lopez@escuela.com	2001-05-12
2	Pedro Martínez	pedro.martinez@escuela.com	2000-08-23
3	Lucía Fernández	lucia.fernandez@escuela.com	2002-11-30

Sentencia SELECT

Pero…
¿y si quiero simplemente los nombres de los estudiantes y no todos sus datos?

Así como utilizamos el asterisco "*" para seleccionar todos los campos de una tabla, podemos también seleccionar los campos en **específico** que deseamos:

```
SELECT nombre FROM estudiantes;
```

Dándonos como resultado:

nombre
María López
Pedro Martínez
Lucía Fernández

Y así como seleccionamos un solo campo, podemos con **multiples** campos:

```
SELECT nombre, fecha_nacimiento FROM estudiantes;
```

Dándonos como resultado:

nombre	fecha_nacimiento
María López	2001-05-12
Pedro Martínez	2000-08-23
Lucía Fernández	2002-11-30

Filtros: WHERE

Los filtros, como bien dice su nombre, nos servirán para filtrar las respuestas de nuestras peticiones y obtener resultados más **personalizados** dependiendo de nuestras necesidades.
El filtro **WHERE** lo utilizaremos en conjunto con la sentencia SELECT para especificar una **condición** que deben **cumplir** los datos para ser seleccionados:

```
SELECT nombre, fecha_nacimiento
FROM estudiantes
WHERE fecha_nacimiento > '2001-01-01';
```

Seleccionamos los campos **nombre** y **fecha_nacimiento** de la tabla **estudiantes** y con el filtro **WHERE** agregamos una **condición** que solo los registros que lo cumplan, se van a devolver en la respuesta, o sea, todos los estudiantes que nacieron después del primero de enero del 2001.

Dándonos como resultado:

nombre	fecha_nacimiento
María López	2001-05-12
Lucía Fernández	2002-11-30

Filtros: ORDER BY

El filtro **ORDER BY** , como su traducción lo indica, para ordenar el resultado de la consulta de forma **ASC**endente o **DESC**endente según un campo a elección.

```
SELECT nombre, fecha_nacimiento
FROM estudiantes
ORDER BY fecha_nacimiento DESC;
```

Dándonos como resultado:

nombre	fecha_nacimiento
Lucía Fernández	2002-11-30
María López	2001-05-12
Pedro Martínez	2000-08-23

Además, se pueden personalizar aún más las consultas utilizando ambos filtros al mismo tiempo.

```
SELECT nombre, fecha_nacimiento
FROM estudiantes
WHERE fecha_nacimiento > '2001-01-01'
ORDER BY fecha_nacimiento DESC;
```

Dándonos como resultado:

nombre	fecha_nacimiento
Lucía Fernández	2002-11-30
María López	2001-05-12

¡Los valores que se pueden definir en el filtro **ORDER BY** son **ASC** para **ascendente** y **DESC** para **descendente**!

Joins

Repasemos la estructura de nuestra base de datos. Además de la tabla *estudiantes* tenemos una tabla ***profesores*** y otra tabla ***cursos*** en la cual cada curso tiene un profesor **designado**.
En otras palabras, *cursos* tiene una **foreign key** que hace referencia a la **primary key** de un profesor en la tabla *profesores*.

Imaginemos la siguiente situación:
Los profesores necesitan saber qué curso deben dictar este cuatrimestre y nos piden a nosotros que pasemos la lista de los profesores con sus cursos designados.
Si hacemos una consulta a la tabla de cursos nos devolverá lo siguiente:

id_curso	nombre_curso	descripcion	id_profesor
1	Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	1
2	Literatura Española	Estudio de la literatura de España	2
3	Programación en Python	Introducción a la programación en Python	3

Darles una lista con un ID del profesor no es muy útil, ¿no?
¿Qué sería lo ideal entonces?

Buscaríamos una manera de **combinar** y **vincular** la foreign key de la tabla *cursos* con su respectiva primary key de la tabla *profesores*.
SQL pensó lo mismo que nosotros, y para eso utilizamos ***JOIN***.

Los tipos más **comunes** de *JOIN* son:

- ➔ **INNER JOIN**
- ➔ **LEFT JOIN**
- ➔ **RIGHT JOIN**

Veamos los con ejemplos 👇

INNER JOIN

La característica fundamental que distingue **INNER JOIN** de los demás es que, al combinar dos tablas, solo devolverá los registros que tienen **coincidencias entre sí**.

Primero veamos los datos que tenemos disponible.

Tabla ***profesores***:

id_profesor	nombre	email	telefono
1	Juan Pérez	juan.perez@escuela.com	123-456-7890
2	Ana Gómez	ana.gomez@escuela.com	098-765-4321
3	Carlos Ruiz	carlos.ruiz@escuela.com	555-555-5555
4	Infor Matorio	infor.matorio@escuela.com	635-424-1240

Tabla ***cursos***:

id_curso	nombre_curso	descripcion	id_profesor
1	Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	1
2	Literatura Española	Estudio de la literatura de España	2
3	Programación en Python	Introducción a la programación en Python	3
4	Desarrollo Web	2da etapa Base de datos	NULL

Existen **dos** cosas importantes a tener en cuenta:

- ➔ Hay 4 profesores y solo 3 tienen cursos asignados, es decir, en un curso, NO va a haber un profesor.
- ➔ Un curso NO tiene foreign key y aparece **null** en el campo de *id_profesor*, es decir, que no tiene ningún profesor designado.

Como dijimos anteriormente, INNER JOIN devolverá los registros que tengan **coincidencias entre sí**.

Veamos cómo es la **estructura** para implementarlo:

```
SELECT columnas
FROM tabla1 INNER JOIN tabla2
ON tabla1.columna = tabla2.columna
```

INNER JOIN

Con todos los conceptos vistos, veamos un ejemplo:

```
SELECT *
FROM cursos INNER JOIN profesores
ON cursos.id_profesor = profesores.id_profesor
```

Primero seleccionamos todas las columnas con **"*"** de ambas tablas.
Luego desde la tabla *cursos,* **unimos** a la tabla *profesores* utilizando **INNER JOIN**.
Por último, usamos la palabra reservada ***ON*** para especificar la **condición** que define cómo se deben combinar los registros. En nuestro ejemplo queremos que se combinen si el **ID** del profesor coinciden en ambas tablas.
Nos da como resultado esto:

id_curso	nombre_curso	descripcion	id_profesor	id_profesor	nombre	email	telefono
1	Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	1	1	Juan Pérez	juan.perez@escuela.com	123-456-7890
2	Literatura Española	Estudio de la literatura de España	2	2	Ana Gómez	ana.gomez@escuela.com	098-765-4321
3	Programación en Python	Introducción a la programación en Python	3	3	Carlos Ruiz	carlos.ruiz@escuela.com	555-555-5555

Nótese que el curso que **no** tenía un profesor designado y el profesor que tampoco estaba designado en ningún curso, ahora aparecen, esto es debido a que el *INNER JOIN* solo muestran **coincidencias entre tablas** y el resto lo descarta.

Por último, nos damos cuenta que el resultado sigue sin ser lo suficientemente pulcro.
Tiene datos **redundantes** y otros que son **innecesarios** para la petición que nos hicieron los profesores.

Veamos cómo podemos arreglarlo 👉

INNER JOIN

Como ya aprendimos al comienzo de esta clase, podemos seleccionar los campos que deseamos con la sentencia **SELECT**.
Por lo tanto, debemos modificar el **"*"** con los campos que queremos que se muestren en el resultado:

```
SELECT nombre_curso AS Curso, descripcion, nombre AS Profesor
FROM cursos INNER JOIN profesores
ON cursos.id_profesor = profesores.id_profesor
```

En este ejemplo, además de elegir las columnas que se van a mostrar, ocupamos la palabra reservada ***AS*** que nos sirve para darle un alias a la columna debido a que ***nombre*** y ***nombre_curso*** no eran muy descriptivos una vez fuera de su tabla original sin su contexto.

Dándonos como resultado la siguiente tabla:

Con el uso de **AS**:

Curso	descripcion	Profesor
Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	Juan Pérez
Literatura Española	Estudio de la literatura de España	Ana Gómez
Programación en Python	Introducción a la programación en Python	Carlos Ruiz

Sin el uso de **AS:**

nombre_curso	descripcion	nombre
Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	Juan Pérez
Literatura Española	Estudio de la literatura de España	Ana Gómez
Programación en Python	Introducción a la programación en Python	Carlos Ruiz

LEFT JOIN y RIGHT JOIN

LEFT JOIN

La operación **LEFT JOIN** tiene la particularidad de que devolverá todos los registros de la *tabla de la izquierda* (LEFT).
Los registros que tengan coincidencia con la otra tabla en cuestión, se **combinaran** al igual que *INNER JOIN*, pero los registros que **no**, se rellenaran con el valor **null.**
La estructura es la misma que la anterior, solo cambiamos **INNER** por **LEFT**:

```
SELECT nombre_curso, descripcion, nombre
FROM cursos LEFT JOIN profesores
ON cursos.id_profesor = profesores.id_profesor
```

nombre_curso	descripcion	nombre
Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	Juan Pérez
Literatura Española	Estudio de la literatura de España	Ana Gómez
Programación en Python	Introducción a la programación en Python	Carlos Ruiz
Desarrollo Web	2da etapa Base de datos	NULL

RIGHT JOIN

La operación **RIGHT JOIN** es similar a su contraparte **LEFT JOIN**, con la diferencia de que devolverá todos los registros de la *tabla derecha* y las que **no** coincidan con la tabla de la izquierda se rellenaran con **null**

```
SELECT nombre_curso, descripcion, nombre
FROM cursos RIGHT JOIN profesores
ON cursos.id_profesor = profesores.id_profesor
```

nombre_curso	descripcion	nombre
Matemáticas Avanzadas	Curso de matemáticas para nivel avanzado	Juan Pérez
Literatura Española	Estudio de la literatura de España	Ana Gómez
Programación en Python	Introducción a la programación en Python	Carlos Ruiz
NULL	NULL	Infor Matorio

Así que, como en **LEFT JOIN** el curso que no tenía un profesor designado se lo devolvió igual, pero con un campo **null**.
Por otro lado, con **RIGHT JOIN** se devolvió todos los profesores pero con el valor **null** en el profesor sin curso designado 😌

3

SQL Avanzado

Ya aprendimos las sentencias y cláusulas fundamentales para el manejo de base de datos con SQL.

Pero como todo en el mundo de la programación, siempre se puede profundizar y complejizar cada vez más, veamos que nuevas funcionalidades nos provee SQL.

Subconsultas

Estructura de la subconsulta

Las subconsultas se utilizan para realizar operaciones más **complejas anidando consultas dentro de otras consultas**, como obtener datos que dependen de otros datos ya procesados previamente.

La estructura es la siguiente:

```
SELECT columna1
FROM tabla
WHERE columna = ( SELECT columna FROM otra_tabla WHERE condicion );
```

En esta estructura, tenemos en la primer cláusula **WHERE** una condición donde se busca la igualdad entre *columna* y la *subconsulta* que se encuentra entre **paréntesis.**

En orden de resolución primero se resuelve la **subconsulta** (la consulta que se encuentra entre paréntesis) y una vez que devuelva ese resultado, *columna* realiza la **comparación** de igualdad con dicho resultado.

Veamos con un ejemplo 👇

Subconsultas

Si quisiéramos saber cual es el curso con más estudiantes inscriptos en la base de datos, podríamos dividir la operación en **dos partes:**

1. **Contabilizar** la cantidad de inscriptos por curso y filtrar el mayor.
2. Buscar el curso resultante por su **ID** en la tabla de cursos.

```sql
SELECT nombre_curso
FROM cursos
WHERE id_curso = (
    SELECT id_curso
    FROM inscripciones
    GROUP BY id_curso
    ORDER BY COUNT(id_estudiante) DESC
    LIMIT 1
);
```

👇

nombre_curso
Programación en Python

Dentro de la subconsulta utilizamos el operador **GROUP BY** para agrupar todos los registros de una columna específica (id_curso) que tengan el mismo valor.
Es decir, que si en nuestra base de datos está repetido 3 veces el mismo *id_curso*, se genera un grupo con esos 3 registros. Y así con cada *id_curso*.

En base a esa agrupación, puedo utilizar el operador aritmético **COUNT()** que contará la cantidad de cursos por grupo creado.
Por último, **LIMIT 1** limitará que el resultado solo devuelva el primer valor, es por eso que utilizamos **ORDER BY DESC** para que el mayor quede primero y sea el que devuelva.

Una vez terminada la subconsulta, se ejecutará la consulta principal, que es, buscar en la tabla *cursos* el *id_curso* obtenido de la consulta anterior.

Funciones agregadas

SUM, COUNT, AVG...

Las funciones agregadas las utilizamos para realizar **cálculos** y devolver un **único** valor como respuesta.
Entre los más utilizados tenemos:

- ➔ **SUM** (suma de valores)
- ➔ **COUNT** (conteo de filas)
- ➔ **AVG** (promedio de valores)

Para operar con estas funciones, para un grupo específico de datos, en lugar de para toda la tabla, ocuparemos **GROUP BY** que vimos en el ejemplo anterior de subconsultas para agrupar los registros que tienen el mismo valor en una columna seleccionada, para luego poder aplicar la función agregada que deseemos en cada grupo.

Estructura:

```
SELECT funcion_agregada(columna)
FROM tabla
WHERE condicion
GROUP BY columna;
```

.

En caso de que queramos aplicar estas funciones a **todos los datos** de nuestra tabla no va a ser necesario.

Estructura:

```
SELECT funcion_agregada(columna)
FROM tabla;
```

.

Funciones agregadas

SUM()

Actualizamos la base de datos y ahora tenemos el campo **duracion_meses** en la tabla ***cursos*** en el cual tenemos el tiempo que dura (en meses) cada curso.

Si queremos saber cuánto tiempo nos llevaría hacer todos los cursos de la universidad, podemos utilizar la función **SUM()** que sumará todos los valores de todos los registros del campo específico.

```
SELECT SUM(duracion_meses) AS total_duracion_meses
FROM cursos;
```

👇

total_duracion_meses
38

AVG()

Por otro lado, si quisiéramos saber el promedio de duración que conllevan los cursos de la universidad, podemos utilizar la función **AVG()**. Esta nos permite calcular el promedio de todos los valores de una columna.

```
SELECT AVG(duracion_meses) AS promedio_duracion_meses
FROM cursos;
```

.

👇

promedio_duracion_meses
4.7500

Nos da un promedio de 4.75 meses de duración por cada curso de la universidad.

Transacciones y control de concurrencia

Transacciones

Las **transacciones** y el **control de concurrencia** son conceptos importantísimos en base de datos para proteger la integridad y consistencia de los datos.

Una transacción es una secuencia de operaciones SQL que va a garantizar que todas las operaciones dentro de la misma se completen con **éxito** o que **no se aplique** en caso contrario.
Las propiedades fundamentales de las transacciones se conocen como **ACID** *(Atomicity Consistency Isolation Durability)*:

- **Atomicidad (Atomicity):**
  - Si alguna parte de la transacción falla, entonces toda la transacción vuelve a su estado inicial. Esto asegura que las operaciones no queden en un estado intermedio.
- **Consistencia (Consistency):**
  - La consistencia asegura que las transacciones no violen la integridad de los datos haciéndolo pasar de un estado válido a otro estado válido siempre y cuando la base de datos cumpla con todas las **reglas** y **restricciones** antes y después de la transacción.
- **Aislamiento (Isolation):**
  - El aislamiento garantiza que las transacciones se ejecuten de forma **independiente** y no afecten los resultados de otras transacciones en curso.
- **Durabilidad (Durability):**
  - Una vez que una transacción se ha confirmado con un ***COMMIT***, sus cambios persistirán en la base de datos, incluso si ocurre una **falla**, los cambios hechos no se perderán.

Ejemplo: Transacción

En este ejemplo agregaremos un nuevo registro en la tabla *inscripciones* y luego verificaremos que se encuentre esa inscripción en la tabla.

Utilizamos las palabras reservadas **START TRANSACTION** para iniciar una nueva transacción. A partir de esa línea, cualquier cambio que se haga en la base de datos se mantendrá en un estado *transaccional* hasta que se **confirme** o se **revierta**.

Luego agregamos el nuevo registro a la tabla inscripciones con **INSERT INTO.**

Verificamos que la inscripción haya impactado en la tabla con **SELECT** filtrando por *id_estudiante*.

Y por último, utilizamos **COMMIT;** para confirmar que la transacción salió correctamente y se guarden los cambios en la base de datos permanentemente.

```
START TRANSACTION;

INSERT INTO inscripciones (id_estudiante, id_curso, fecha_inscripcion)
VALUES (4, 6, '2024-04-01');

SELECT * FROM inscripciones WHERE id_estudiante = 4;

COMMIT;
```

👇

id_inscripcion	id_estudiante	id_curso	fecha_inscripcion
6	4	2	2024-02-10
7	4	3	2024-02-15
9	4	6	2024-04-01
NULL	NULL	NULL	NULL

Así como existe **COMMIT;** para guardar los cambios en base de datos, en caso de haber algún problema, también podemos utilizar **ROLLBACK;** para que restaure los cambios a su versión anterior previo al conflicto.

```
ROLLBACK;
```

Transacciones y control de concurrencia

Control de concurrencia

El control de concurrencia es un conjunto de técnicas que utilizamos para gestionar el acceso simultáneo a los datos en el caso de que haya múltiples transacciones.

Con estas técnicas buscamos garantizar la consistencia y la integridad de la base de datos, evitando problemas como:

- **Lecturas sucias:**
  - Ocurre cuando una transacción lee datos que todavía no fueron confirmados en otras transacciones.
- **Lecturas no repetibles**:
  - Cuando una transacción lee los mismos datos más de una vez pero sus valores cambian porque otra transacción lo está modificando.
- **Fantasmas**:
  - Se debe a que una transacción ve un conjunto de datos que cambia porque otra transacción está agregando o eliminando datos.

Ejemplo: Control de concurrencia

Transacción 1

```
START TRANSACTION;

SELECT * FROM inscripciones WHERE id_estudiante = 1 FOR UPDATE;

INSERT INTO inscripciones (id_estudiante, id_curso, fecha_inscripcion)
VALUES (1, 8, '2024-04-15');

COMMIT;
```

Transacción 2

```
START TRANSACTION;

INSERT INTO inscripciones (id_estudiante, id_curso, fecha_inscripcion)
VALUES (1, 9, '2024-04-16');

COMMIT;
```

Supongamos que dos personas desean modificar el mismo registro en la base de datos en **simultáneo**.
En la **transacción 1**, en la sentencia **SELECT** el usuario bloqueó el acceso a dicho registro utilizando la sentencia **FOR UPDATE.**
Es decir que, ese registro en particular **no** puede ser modificado ni accedido hasta que la transacción 1 finalice (se realice el **COMMIT** o **ROLLBACK**).

En resumen, cuando el usuario que realizó la transacción 2 intenta modificar ese mismo registro, quedará en un estado de bloqueo hasta que finalice la transacción 1.
Una vez finalizado, actualizará los valores y continuará con su ejecución.

Sí transacción 2 intentase modificar otro registro de la misma tabla, no habría conflicto y puede continuar sin esperar.

Lo que vimos hoy 😉

Terminamos nuestro proceso de diseño aprendiendo a interpretar y transformar nuestro diagrama entidad-relación a tablas listas para su implementación.

Además nos adentramos levemente con SQL, aprendiendo la manipulación básica de los datos, los modos de respuesta y técnicas fundamentales para proteger la integridad de nuestros datos y futuros posibles conflictos.

¡Nos vemos En la próxima clase!