

**Universidad Nacional del Nordeste**



**Facultad de Ciencias Exactas y Naturales y Agrimensura**

**Carrera**: Licenciatura en Sistemas de Información

**Asignatura:** Base de Datos I

**Tema:** Manejo de tipos de datos JSON

**Año:** 2023

**GRUPO 6**

**Profesor:** Darío Villegas

**Alumnos:**

* Fernández Rolón, Sergio Gabriel
* Gómez, Daniel Fernando

INDICE

[Capítulo I: Introducción 3](#_Toc151419333)

[Tema 3](#_Toc151419334)

[Planteamiento *del* problema 4](#_Toc151419335)

[Objetivos Específicos 4](#_Toc151419336)

[Capitulo II: Marco conceptual o referencial 4](#_Toc151419337)

[**JSON (JavaScript Object Notation):** 4](#_Toc151419338)

[Integración **de JSON en MySQL:** 5](#_Toc151419339)

[Operaciones **CRUD con JSON:** 5](#_Toc151419340)

[**Consultas JSON en MySQL:** 5](#_Toc151419341)

[Indexación **en MySQL:** 5](#_Toc151419342)

[**Consideraciones de Rendimiento:** 5](#_Toc151419343)

[CapituloIII:Metodología seguida 6](#_Toc151419344)

[FUNCIONES PARA MANEJO DE CAMPOS EN TIPO JSON 6](#_Toc151419345)

[JSON\_OBJECT 6](#_Toc151419346)

[JSON\_REPLACE 7](#_Toc151419347)

[JSON\_REMOVE 7](#_Toc151419348)

[JSON\_INSERT 8](#_Toc151419349)

[JSON\_VALID 8](#_Toc151419350)

[JSON\_EXTRACT 9](#_Toc151419351)

[JSON\_UNQUOTE 9](#_Toc151419352)

[Querys 10](#_Toc151419353)

[VISTAS 11](#_Toc151419354)

[Diferencias entre vistas y vistas materializadas 14](#_Toc151419355)

# Capítulo I: Introducción

El manejo de tipos de datos JSON en bases de datos es un aspecto crítico en el desarrollo de aplicaciones modernas. Con la creciente popularidad de JSON como formato de intercambio de datos, es esencial comprender cómo almacenar, consultar y optimizar datos JSON en bases de datos relacionales. En este trabajo, exploraremos los desafíos y las mejores prácticas asociadas con esta tarea, brindando información valiosa para desarrolladores y profesionales de bases de datos.

## Tema

Este Trabajo Práctico se enfoca en la gestión de tipos de datos JSON en el contexto de bases de datos relacionales. Con el auge de aplicaciones web y móviles, el uso de JSON como formato de intercambio de datos se ha vuelto común. Sin embargo, integrar eficientemente estos tipos de datos en bases de datos relacionales plantea desafíos particulares. El objetivo de este trabajo es investigar cómo se pueden almacenar, consultar y optimizar los tipos de datos JSON dentro de sistemas de bases de datos relacionales.

El trabajo explorara las siguientes áreas:

1 – **Integración de JSON en Bases de Datos Relacionales:** Una introducción a cómo se pueden almacenar datos JSON en tablas de bases de datos relacionales y cómo se relacionan con otros tipos de datos.

2 - **Consultas y búsqueda de Datos JSON:** Cómo realizar consultas efectivas en datos JSON, incluyendo la extracción de información específica de objetos y arrays anidados.

3 - **Indexación y Optimización:** Estrategias para indexar y optimizar el rendimiento de consultas en datos JSON, considerando la escalabilidad y eficiencia.

4 - **Transacciones y Mantenimiento:** Consideraciones sobre la transaccionalidad y el mantenimiento de datos JSON en entornos de bases de datos relacionales.

5 – **Ejemplo de casos de uso:** Estudios de casos prácticos que demuestran cómo el manejo de datos JSON en bases de datos puede ser beneficioso en aplicaciones del mundo real.

Este trabajo proporcionará una visión detallada sobre cómo se pueden aprovechar los tipos de datos JSON en bases de datos relacionales, abordando los desafíos y las mejores prácticas para su implementación y optimización. Los estudiantes y profesionales interesados en la integración de datos JSON en bases de datos encontrarán valiosa información para mejorar sus habilidades y conocimientos en este campo específico.

## Planteamiento *del* problema

El problema central que este Trabajo Práctico busca abordar es el siguiente:

¿Cómo se pueden integrar eficazmente tipos de datos JSON en bases de datos relacionales, garantizando un almacenamiento eficiente, la capacidad de consulta y la optimización del rendimiento?

El auge de aplicaciones web y móviles ha llevado a un aumento en la utilización de JSON como formato de intercambio de datos. Sin embargo, la gestión adecuada de datos JSON en bases de datos relacionales plantea interrogantes fundamentales. Estos cuestionamientos abarcan desde la estructura de datos y su almacenamiento hasta la capacidad de realizar consultas efectivas y mantener un rendimiento óptimo en entornos de producción.

## Objetivos Específicos

**Analizar la estructura de datos JSON:** Comprender en detalle la estructura de datos JSON, incluyendo objetos, arrays y valores, y cómo se relacionan con las tablas de bases de datos relacionales.

**Explorar la capacidad de consulta:** Investigar las técnicas y herramientas disponibles para realizar consultas efectivas en datos JSON, incluyendo la extracción de información específica de objetos y arrays anidados.

**Optimizar el rendimiento:** identificar y evaluar prácticas de optimización de rendimiento para consultas que involucran datos JSON en bases de datos relacionales.

Estudiar ejemplos de casos de uso: Presentar ejemplos de casos de uso prácticos que demuestren la aplicación efectiva de la gestión de datos JSON en entornos del mundo real.

# Capitulo II: Marco conceptual o referencial

**JSON (JavaScript Object Notation):** es un formato ligero de intercambio de datos que representa objetos y estructuras de datos legibles por humanos. Está compuesto por pares clave-valor y puede incluir arrays y objetos anidados.

## Integración **de JSON en MySQL:**

Tipo de Datos JSON: MySQL proporciona un tipo de datos específico llamado JSON para almacenar datos en formato JSON. Este tipo de datos permite almacenar y manipular datos semiestructurados.

## Operaciones **CRUD con JSON:**

**Inserción de Datos JSON:**

**INSERT INTO ejemplo** (id, datos\_json) **VALUES** (1, '{"nombre": "Ejemplo", "edad": 25}'

**Consulta y Modificación de Datos JSON:**

**SELECT** datos\_json->'$.nombre' **AS** nombre **FROM** ejemplo **WHERE** id = 1;

**UPDATE** ejemplo **SET** datos\_json = '{"nombre": "Nuevo Ejemplo", "edad": 30}' **WHERE** id = 1;

## **Consultas JSON en MySQL:**

**Operadores y Funciones JSON:**

MySQL proporciona operadores y funciones (JSON\_EXTRACT, JSON\_UNQUOTE) específicas para trabajar con datos JSON.

## Indexación **en MySQL:**

**Índices Normales vs. Índices JSON:**

Puedes crear índices normales en columnas JSON para acelerar las búsquedas.

**CREATE INDEX** idx\_nombre **ON** ejemplo((datos\_json->'$.nombre'));

Los índices JSON pueden ser útiles para optimizar la búsqueda de claves específicas dentro de los datos JSON.

**CREATE INDEX** idx\_edad **ON** ejemplo((datos\_json->'$.edad'));

## **Consideraciones de Rendimiento:**

**Volumen de Datos e Indexación:**

Evalúa el volumen de datos y el impacto de la indexación en el rendimiento.

La indexación selectiva puede mejorar la eficiencia de las consultas.

# CapituloIII:Metodología seguida

* Creamos la base de datos.
* Descargamos y corregimos algunos de los Script de Comercio. SQL, los convertimos a formato MySQL y los ejecutamos.
* Importamos el lote de datos.
* Generamos el diagrama entidad – relación.

## FUNCIONES PARA MANEJO DE CAMPOS EN TIPO JSON

En este caso, insertamos un registro con el campo apeynom en formato JSON, que tiene la siguiente estructura:

{

“APELLIDO”: VALOR,

“NOMBRE”: VALOR

}

**INSERT INTO** base\_consorcio.conserje (apeynom, tel, fechnac, estciv)

**VALUES** ('*{"APELLIDO": "GAUNA", "NOMBRE": "PEDRO"}*', 3794651348, '1998-12-12', 'S');

### JSON\_OBJECT

Esta función se utiliza para construir un objeto JSON a partir de una lista de pares CLAVE-VALOR. Esta función toma un numero variable de argumentos, donde cada parámetro debe ser un par CLAVE-VALOR. La función genera un objeto JSON que tiene las claves y valores proporcionados.

En el siguiente ejemplo insertamos un registro en la tabla conserje, donde en el campo **apeynom**, insertamos un objeto json, utilizando la función JSON\_OBJECT.

**INSERT INTO** base\_consorcio.conserje (apeynom, tel, fechnac, estciv)

**VALUES** (**JSON\_OBJECT**('APELLIDO', 'ACUÑA', 'NOMBRE', 'JUAN'), 3497615841, '1990-12-21', 'C');

EJEMPLO CON SELECT

**SELECT**

cons.idconserje,

**JSON\_OBJECT**('APELLIDO Y NOMBRE', apeynom) AS nombre,

**JSON\_OBJECT**('TELEFONO', tel) AS telefono,

cons.fechnac,

**JSON\_OBJECT**('ESTADO CIVIL', CASE WHEN estciv = 'S' THEN 'SOLTERO' WHEN estciv = 'C' THEN 'CASADO' END) AS estado\_civil

**FROM** base\_consorcio.conserje AS cons;

**SELECT**

cons.idconserje,

**JSON\_OBJECT**('APELLIDO Y NOMBRE', apeynom,

'TELEFONO', tel,

'ESTADO CIVIL', CASE WHEN estciv = 'S' THEN 'SOLTERO' WHEN estciv = 'C' THEN 'CASADO' END) AS conserje

**FROM** base\_consorcio.conserje AS cons;

### JSON\_REPLACE

Esta función se utiliza para reemplazar valores de un objeto **JSON**. La función toma tres argumentos:

1 – **Columna:** Es la columna que contiene el objeto **JSON** en el que se desea realizar el reemplazo.

2 – **Ruta:** Especifica la ubicación del valor que se desea reemplazar dentro del objeto **JSON**.

3 – **Nuevo Valor:** Es el valor que se desea insertar en la ubicación especificada en la ruta.

Teniendo como ejemplo el registro que insertamos anteriormente, el **idconserje** asignado fue el 222, en este caso modificamos el NOMBRE, reemplazando PEDRO por SERGIO.

**UPDATE** base\_consorcio.conserje

**SET** apeynom = **JSON\_REPLACE**(apeynom, '$.NOMBRE', 'SERGIO')

**WHERE** idconserje = 222;

### JSON\_REMOVE

Esta función se utiliza para eliminar elementos de un objeto JSON, Se puede usar esta función para eliminar un objeto especifico o varios elementos de un objeto JSON. La función toma dos argumentos:

1 – **Columna:** Es la columna que contiene el objeto JSON del cual se desea eliminar elementos.

2 – **Ruta:** Especifica la ubicación del elemento que se desea eliminar dentro del objeto JSON.

Siguiendo con el mismo registro que venimos trabajando, en este caso removemos el elemento NOMBRE de la columna apeynom.

**UPDATE** base\_consorcio.conserje

**SET** apeynom =  **JSON\_REMOVE**(apeynom, '$.NOMBRE')

**WHERE** idconserje = 222;

Donde sí después hacemos un select de este registro, la columna apeynom, devuelve lo siguiente

'*{"APELLIDO": "GAUNA"}*

### JSON\_INSERT

Esta función se utiliza para insertar un nuevo elemento en un objeto JSON. Se puede utilizar para agregar un elemento en una posición especifica dentro del objeto. La función toma tres argumentos.

1 – **Columna JSON:** Es la columna que contiene el objeto JSON en la cual se desea insertar un nuevo elemento.

2 – **Ruta:** Especifica la ubicación donde se desea insertar el nuevo elemento dentro del objeto JSON. Se puede usar esta ruta para navegar por la estructura del objeto y determinar la posición de inserción.

3 – **Nuevo Valor:** Es el valor que se desea insertar en la posición especificada por la ruta.

Trabajando sobre el mismo registro, en este caso volvemos a insertar el elemento NOMBRE con el valor PEDRO.

**UPDATE** base\_consorcio.conserje

**SET** apeynom = **JSON\_INSERT**(apeynom, '$.NOMBRE', 'PEDRO')

**WHERE** idconserje = 222;

### JSON\_VALID

Esta función se utiliza para validar si una cadena de texto representa un documento JSON valido. La función toma un solo argumento, que es la cadena que se quiere validar como JSON:

Utilizando la base de datos de consorcio, mantuvimos los registros que nos enviaron, y fuimos insertando nuevos registros con algunos de los campos en formato JSON, en la siguiente consulta, hacemos un simple SELECT, pero en la cláusula WHERE especificamos que traiga los registros donde campo apeynom cumplan con la estructura tipo JSON

**SELECT \* FROM** base\_consorcio.conserje

**WHERE JSON\_VALID**(apeynom);

### JSON\_EXTRACT

Esta función se utiliza para extraer un valor especifico de un campo con estructura JSON. Se puede usar esta función para recuperar un valor particular en función de una ruta proporcionada. La función toma dos argumentos:

1 – Columnas JSON: Es la columna que contiene el objeto JSON del cual se desea extraer un valor.

2 – Ruta: Especifica la ubicación del valor que se desea extraer dentro del objeto JSON.

**SELECT**

**JSON\_EXTRACT**(apeynom, '$.NOMBRE') AS nombre

**FROM** base\_consorcio.conserje

**WHERE** idconserje = 222;

En este caso, nos retorna un registro con el siguiente valor “PEDRO”, con las comillas incluidas, que la podemos eliminar con la siguiente función.

### JSON\_UNQUOTE

Esta función se utiliza para eliminar las comillas que rodean una cadena JSON. Esta función toma un solo argumento, que es la cadena que contiene un valor JSON rodeado de comillas.

Con solo agregar esta función a la consulta anterior, eliminamos las comillas.

**SELECT**

**JSON\_UNQUOTE**(**JSON\_EXTRACT**(apeynom, '$.NOMBRE')) AS nombre

**FROM** base\_consorcio.conserje

**WHERE** idconserje = 222;

Fuimos insertando nuevos registros en las distintas tablas de la base datos, con alguno de los campos en formato JSON.

### Querys

INSERT INTO `base\_consorcio`.`conserje` (`apeynom`, `tel`, `fechnac`, `estciv`)

VALUES ('{\"nombre:\":\"GABRIEL\", \"apellido\":\"FERNANDEZ\"}', '3794701105', '1996-11-29', 'S');

UPDATE `base\_consorcio`.`conserje` SET `apeynom` = '{\"APELLIDO\":\"FERNANDEZ\", \"NOMBRE\":\"GABRIEL\"}' WHERE (`idconserje` = '221');

INSERT INTO `base\_consorcio`.`administrador` (`idadmin`, `apeynom`, `viveahi`, `tel`, `sexo`, `fechnac`)

VALUES ('175', '{\"APELLIDO\":\"GOMEZ\",\"NOMBRE\":\"FERNANDO\"}', 'S', '3658962135', 'M', '1996-09-23');

INSERT INTO `base\_consorcio`.`consorcio` (`idprovincia`, `idlocalidad`, `idconsorcio`, `nombre`, `direccion`, `idzona`, `idconserje`, `idadmin`)

VALUES ('7', '7', '2', 'EDIFICIO-7011', '{\"CALLE\":\"TACUARI\", \"N°\":\"1316\"}', '1', '221', '175');

Consulta para extraer la tabla principal que es la de consorcio, con todos los datos relacionados a la misma, utilizando en este caso, algunos campos de las tablas como dato de tipo JSON.

**SELECT**

cons.nombre **AS** nombre\_consorcio,

IFNULL(**JSON\_UNQUOTE**(JSON\_EXTRACT(cons.direccion,'$.CALLE')), cons.direccion) **AS** calle,

**JSON\_UNQUOTE**(JSON\_EXTRACT(cons.direccion, '$.N°')) **AS** numero,

prov.descripcion **AS** nombre\_provincia,

loc.descripcion **AS** nombre\_localidad,

CONCAT(**JSON\_UNQUOTE**(JSON\_EXTRACT(conser.apeynom, '$.NOMBRE')), ', ',JSON\_UNQUOTE(JSON\_EXTRACT(conser.apeynom, '$.APELLIDO') )) **AS** nombre\_conserje,

CONCAT(**JSON\_UNQUOTE**(JSON\_EXTRACT(admin.apeynom, '$.NOMBRE')), ', ', JSON\_UNQUOTE(JSON\_EXTRACT(admin.apeynom, '$.APELLIDO') ) ) **AS** nombre\_administrador

**FROM** base\_consorcio.consorcio AS cons

**INNER JOIN** base\_consorcio.administrador AS admin

**ON** cons.idadmin = admin.idadmin

**INNER** **JOIN** base\_consorcio.provincia AS prov

**ON** cons.idprovincia = prov.idprovincia

**INNER** **JOIN** base\_consorcio.localidad AS loc

**ON** cons.idlocalidad = loc.idlocalidad

**AND** loc.idprovincia = prov.idprovincia

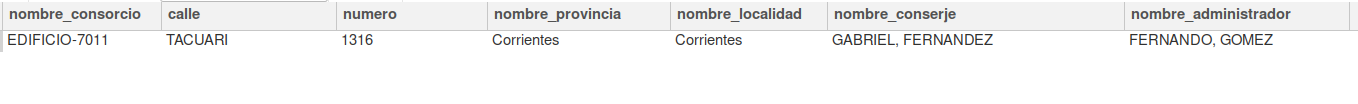
**INNER** **JOIN** base\_consorcio.conserje AS conser

**ON** cons.idconserje = conser.idconserje

**WHERE** cons.idadmin = 175;



Esta consulta nos devuelve lo siguiente:



Si bien parece una consulta simple, los registros tienen la siguiente estructura:

### VISTAS

Creamos la vista **provincia\_localidad** donde hacemos un inner join de 2 tablas, la de **localidad** y **provincia,** esta vista nos permite hace un solo join en lugar de tener que hacer 2 join con las dos tablas.

**CREATE VIEW** provincia\_localidad **AS**

**SELECT**

loc.idprovincia,

prov.descripcion AS nombre\_provincia,

loc.idlocalidad,

loc.descripcion AS nombre\_localidad

**FROM** base\_consorcio.localidad **AS** loc

**INNER JOIN** base\_consorcio.provincia **AS** prov

**ON** loc.idprovincia = prov.idprovincia;

Creamos la vista consorcios\_por\_zona, donde agregamos una columna a la tabla de zonas, con la cantidad de consorcios en cada una.

**CREATE VIEW** consorcios\_por\_zona AS

**SELECT**

z.\*,

count(\*) AS total\_consorcios

**FROM** zona AS z

**INNER JOIN** consorcio AS c

**ON** z.idzona = c.idzona

**GROUP BY** z.idzona

**ORDER BY** **COUNT**(\*) **DESC**;

La siguiente consulta utilizamos la vista creada anteriormente para traer el nombre de la provincia y localidad:

**SELECT**

cons.nombre,

cons.direccion,

vista.nombre\_provincia,

vista.nombre\_localidad

**FROM** base\_consorcio.consorcio **AS** cons

**INNER** **JOIN** base\_consorcio.provincia\_localidad **AS** vista

**ON** cons.idprovincia = vista.idprovincia

**AND** cons.idlocalidad = vista.idlocalidad;

La misma consulta sin utilizar la vista

**SELECT**

cons.nombre,

cons.direccion,

prov.descripcion AS provincia,

loc.descripcion AS localidad

**FROM** base\_consorcio.consorcio AS cons

**INNER** **JOIN** base\_consorcio.localidad AS loc

**ON** cons.idlocalidad = loc.idlocalidad

**AND** cons.idprovincia = loc.idprovincia

**INNER** **JOIN** base\_consorcio.provincia AS prov

**ON** loc.idprovincia = prov.idprovincia;

Creamos la siguiente vista para tener el total de gastos de cada consorcio agrupado por tipo de gasto.

**CREATE VIEW** balance\_gastos AS

**SELECT**

gasto.idconsorcio,

gasto.idprovincia,

gasto.idlocalidad,

tipo.descripcion AS gasto,

count(DISTINCT idgasto) AS cantidad,

ROUND(AVG(gasto.importe), 2)AS promedio,

sum(gasto.importe) AS total\_gastos

**FROM** base\_consorcio.gasto

**INNER** **JOIN** base\_consorcio.tipogasto AS tipo

**ON** gasto.idtipogasto = tipo.idtipogasto

**GROUP** **BY** idconsorcio, gasto.idprovincia, gasto.idlocalidad, gasto.idtipogasto;

En la siguiente consulta utilizamos las dos vistas creadas anteriormente, y listamos todos los consorcios con los respectivos totales de gastos, ordenados de forma descendente.

**SELECT**

cons.nombre,

cons.direccion,

vista\_prov.nombre\_provincia,

vista\_prov.nombre\_localidad,

SUM(vista\_gasto.total\_gastos) AS totales

**FROM** base\_consorcio.consorcio AS cons

**INNER** **JOIN** provincia\_localidad AS vista\_prov

**ON** cons.idprovincia = vista\_prov.idprovincia

**AND** cons.idlocalidad = vista\_prov.idlocalidad

**INNER** **JOIN** balance\_gastos AS vista\_gasto

**ON** cons.idconsorcio = vista\_gasto.idconsorcio

**AND** cons.idprovincia = vista\_gasto.idprovincia

**AND** cons.idlocalidad = vista\_gasto.idlocalidad

**GROUP** **BY** cons.idconsorcio, cons.idprovincia, cons.idlocalidad

**ORDER** **BY** totales **DESC**;

### Diferencias entre vistas y vistas materializadas

En las bases de datos relacionales, una vista es una tabla temporal que se crea cuando se transforman y combinan los datos en varias tablas base. Es una tabla virtual que no almacena ningún dato por sí misma. En cambio, se define mediante una consulta en una o más tablas de origen.

Cada vez que un usuario consulta la vista, el motor de base de datos calcula dinámicamente los resultados ejecutando la consulta subyacente en las tablas de origen. Los datos de una vista están siempre actualizados porque se derivan directamente de las tablas de origen cada vez que se accede a ellos.

Una vista materializada, por otro lado, almacena los resultados de una consulta específica como una tabla física en la base de datos. Los datos de la vista materializada se calculan con anterioridad y se almacenan, lo que significa que los resultados ya están disponibles sin necesidad de volver a calcular la consulta cada vez que se accede a la vista.

Sin embargo, los datos de las vistas materializadas no siempre están actualizados. Se debe configurar la frecuencia de la actualización para equilibrar la actualización de los datos y el rendimiento de las consultas.