INSTITUTO POLITECNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA EN INGENIERIA Y TECNOLGIAS AVANZADAS - IPN

MATERIA

Bases de Datos Distribuidas

ALUMNOS

Fernández Guerrero Keb Sebastián

Ramírez Orozco Juan Carlos

Sánchez Herrera Armando Eduardo

PROFESOR

De La Cruz Sosa Carlos

TEMA

Fragmentación, servidores vinculados y particiones en SQL

Grupo 3TM3 Equipo 03 Practica No. 03

Fecha de asignación:	03/06/25 08:30 hrs
Fecha de entrega:	19/06/25 08:30

Introducción: Conceptos usados en la practica

Fragmentación Horizontal Primaria

La fragmentación horizontal primaria (FHP) es una técnica de diseño en bases de datos distribuidas que divide una relación (tabla) en *subconjuntos de tuplas* basados en un predicado definido sobre los atributos de la relación misma. Cada fragmento resultante contiene un grupo de filas que satisfacen una condición específica, usualmente vinculada a las consultas frecuentes o a la localidad de los datos (Özsu & Valduriez, 2020).

Características clave según Özsu:

1. **Predicados de fragmentación**: Se definen condiciones (ej: `WHERE sucursal = "Norte"`) para crear fragmentos.

2. **Completitud**: Cada tupla de la relación original debe pertenecer a algún fragmento.

3. **Reconstrucción**: La unión de todos los fragmentos debe regenerar la relación original sin pérdida de datos.

4. **Disyunción mínima**: Idealmente, los fragmentos no deben superponerse (aunque en la práctica esto depende del diseño).

Ejemplo:

Si tenemos una tabla `EMPLEADOS` y la fragmentamos por región:

- `Fragmento_Emp_Norte`: `WHERE region = 'Norte'`

- `Fragmento_Emp_Sur`: `WHERE region = 'Sur'`

Algoritmos de Derivación de Predicados

La FHP requiere definir predicados lógicos para dividir una tabla. Özsu & Valduriez (2020) describen métodos sistemáticos para derivarlos, especialmente en entornos donde las consultas son complejas o involucran múltiples atributos.

Algoritmo Básico (de Minería de Predicados)

1. Recolección de predicados candidatos:

- Analizar las condiciones (`WHERE`) de las consultas frecuentes.

- Ejemplo: Si las consultas filtran por `departamento = "Ventas"` y `sucursal = "Norte"`, estos son predicados candidatos.

2. Minimización del conjunto de predicados:

- Eliminar redundancias (ej: `sucursal = "Norte" AND sucursal = "Norte"`).

- Combinar predicados compatibles (ej: `departamento = "Ventas" OR departamento = "Marketing"` → `departamento IN ("Ventas", "Marketing") `).

3. Generación de predicados mínimos (Algoritmo de Compleción):

- Asegurar que los predicados sean *completos* (cubran todas las tuplas) y *disjuntos* (no se superpongan, salvo en FHP con superposición controlada).

Ejemplo Práctico:

Para una tabla `PEDIDOS` con consultas frecuentes como:

- `WHERE cliente_region = "Este" AND año = 2023`

- `WHERE cliente_region = "Oeste"

Predicados derivados:

- `P1`: `cliente_region = "Este" AND año = 2023`

- `P2`: `cliente_region = "Oeste"`

- `P3`: `NOT (P1 OR P2)` (fragmento residual para completitud).

Esquemas de Fragmentación Horizontal Primaria

Özsu distingue entre dos enfoques principales:

a) Fragmentación Primaria Directa

- *Definición*: Se aplica directamente a la relación original usando predicados locales.

- *Condiciones:*

  - Los predicados deben ser *relevantes para las consultas* de un sitio específico.

  - Ejemplo: Fragmentar `EMPLEADOS` por `sucursal_id = X` si cada sitio accede principalmente a su propia sucursal.

b) Fragmentación Primaria Derivada

- *Definición*: La fragmentación se basa en predicados de *otra relación relacionada* (vía joins).

- *Ejemplo*:

  - Si `DEPARTAMENTOS` está fragmentada por `region`, entonces `EMPLEADOS` puede fragmentarse derivadamente con:

WHERE EMPLEADOS.depto_id IN (SELECT id FROM DEPARTAMENTOS WHERE region = "Norte")

Reglas Clave:

- *Completitud*: Todos los datos deben estar asignados a algún fragmento (Özsu & Valduriez, 2020).

- *Reconstrucción*: La unión de fragmentos debe igualar la tabla original.

- *Optimización*: Los fragmentos deben minimizar accesos remotos (ej: almacenar datos cercanos a donde se consultan).

Ejemplo Práctico con SQL: Fragmentación Horizontal Primaria

Contexto:

Tenemos una tabla `EMPLEADOS` en una empresa multiregional, con consultas frecuentes filtradas por `region` y `departamento`.

Paso 1: Definir predicados de fragmentación

Analizando las consultas, identificamos:

- Sitio Norte: `WHERE region = 'Norte' AND departamento IN ('Ventas', 'Logística')`.

- Sitio Sur: `WHERE region = 'Sur' AND departamento = 'TI'`.

Paso 2: Crear fragmentos en SQL

```
-- Fragmento para el Norte

CREATE TABLE empleados_norte AS SELECT

* FROM empleados

WHERE region = 'Norte' AND departamento IN ('Ventas', 'Logística');
-- Fragmento para el Sur

CREATE TABLE empleados_sur AS SELECT *

FROM empleados

WHERE region = 'Sur' AND departamento = 'TI';
-- Fragmento residual (opcional, para completitud) CREATE

TABLE empleados_resto AS

SELECT * FROM empleados

WHERE NOT (region = 'Norte' AND departamento IN ('Ventas', 'Logística')) AND

NOT (region = 'Sur' AND departamento = 'TI');
```

Paso 3: Verificar propiedades

- *Completitud*: La unión de los fragmentos reconstruye la tabla original.

- *Disyunción*: Los fragmentos no se superponen (si el diseño es correcto).

Comparación: Fragmentación Primaria (FHP) vs. Derivada (FHD)

Aspecto	Fragmentación Horizontal Primaria (FHP)	Fragmentación Horizontal Derivada (FHD)
Base de la división	Predicados sobre atributos de la tabla misma.	Predicados basados en una tabla relacionada (vía joins).
Ejemplo	WHERE region = 'Norte' (en EMPLEADOS).	WHERE empleado.depto_id IN (SELECT id FROM departamentos WHERE region = 'Norte').
Complejidad	Más simple, directa.	Más compleja, requiere análisis de joins.
Uso típico	Cuando los datos se filtran por atributos locales.	Cuando la fragmentación depende de relaciones entre tablas.
Reconstrucción	Unión de fragmentos.	Unión + joins con la tabla de referencia.
Ventaja	Rendimiento rápido para consultas locales.	Coherencia con datos distribuidos en tablas relacionadas.

Ejemplo de FHD:

Fragmentar `PEDIDOS` basándose en la fragmentación de `CLIENTES`:
-- Asumiendo que CLIENTES está fragmentada por región

CREATE TABLE pedidos_este AS

SELECT p.* FROM pedidos p
JOIN clientes c ON p.cliente_id = c.id WHERE c.region

= 'Este'; ***Conclusión***

- **FHP**: Ideal para datos independientes con acceso localizado.

- **FHD**: Útil cuando la distribución sigue jerarquías (ej: pedidos de clientes por región).

Fuentes Citadas (APA)

- Özsu, M. T., & Valduriez, P. (2020). *Principles of distributed database systems* (4th ed.). Springer.

- Ceri, S., & Pelagatti, G. (1984). *Distributed databases: Principles and systems*. McGraw-Hill.

Partición de Bases de Datos

Concepto:
La partición es una técnica de división de tablas grandes en segmentos más pequeños llamados particiones, que se almacenan y gestionan por separado pero siguen siendo parte de la misma base de datos.

Tipos principales:

1. Partición horizontal (sharding):
    - Divide los registros de una tabla en grupos basados en rangos de valores
    - Ejemplo: Particionar una tabla de pacientes por rangos de ID (1-1000000 en partición 1, 1000001-2000000 en partición 2)
2. Partición vertical:
    - Divide las columnas de una tabla en diferentes grupos
    - Ejemplo: Separar columnas frecuentemente accedidas de las que raramente se consultan

Ventajas de la partición:

- Mejor rendimiento en consultas que acceden solo a particiones específicas
- Operaciones de mantenimiento más eficientes (puedes hacer backup por partición)
- Balanceo de carga al distribuir acceso entre particiones

Servidores Vinculados (Linked Servers)

Concepto:
Los servidores vinculados permiten acceder a datos distribuidos en diferentes servidores de bases de datos como si fueran locales, creando una vista unificada de múltiples fuentes.

Características clave:

- Permiten consultas entre diferentes instancias de SQL Server o incluso diferentes motores (SQL Server → MySQL)
- Usan OPENQUERY o consultas de cuatro partes (servidor.catálogo.esquema.tabla)
- Requieren configuración de seguridad y permisos adecuados

Ejemplo de uso:

SELECT * FROM LINKSERVER.BD.dbo.Tabla

Ventajas:

- Integración transparente de múltiples fuentes de datos

- No requiere replicación de datos para consultas combinadas
- Soporte para transacciones distribuidas (con limitaciones)

Consultas Remotas vs. Consultas Distribuidas en SQL Server

Consultas Remotas

Qué son: Consultas que se ejecutan en un servidor SQL Server pero acceden a datos ubicados en otro servidor remoto.

Características:
- Se conectan a un único servidor remoto
- Usan un nombre de servidor vinculado (linked server)
- Son más simples de configurar
- Ideales para acceder a datos en un solo servidor externo

Consultas Distribuidas

Qué son: Consultas que acceden y combinan datos de múltiples fuentes de datos heterogéneas.

Características:
- Pueden unir datos de varios servidores/sistemas diferentes
- Soportan transacciones distribuidas (MSDTC)
- Más complejas de configurar
- Ideales cuando necesitas combinar datos de múltiples fuentes

Diferencias clave

Característica	Consulta Remota	Consulta Distribuida
Número de fuentes	1 servidor remoto	Múltiples servidores/sistemas
Complejidad	Más simple	Más compleja
Transacciones	No distribuidas	Soporta transacciones distribuidas
Rendimiento	Mejor para accesos simples	Mejor para combinaciones complejas

OpenQuery vs. 4-Part Name

OPENQUERY

*SELECT * FROM OPENQUERY([LINKED_SERVER], 'SELECT * FROM Database.Schema.Table')*

Ventajas:
- La consulta se ejecuta en el servidor remoto
- Mejor para filtrar antes de traer datos
- Soporta sintaxis específica del proveedor remoto

Cuándo usarlo:
- Cuando se quiere que el procesamiento ocurra en el servidor remoto
- Para sistemas heterogéneos (no SQL Server)
- Cuando se necesita usar sintaxis específica del sistema remoto

4-Part Name (Nombre en 4 partes)
SELECT * FROM [LINKED_SERVER].[Database].[Schema].[Table]

Ventajas:
- Sintaxis más simple y familiar
- Mejor integración con el plan de ejecución local
- Más fácil de escribir para consultas simples

Cuando usarlo:
- Para sistemas homogéneos (todos SQL Server)
- Cuando se quiere que SQL Server local optimice la consulta
- Para operaciones sencillas de unión con datos locales

Recomendaciones
Se puede usar consultas remotas cuando:
- Solo se necesita datos de un servidor externo
- No se requiere transacciones distribuidas
- La configuración simple es prioridad

Usa consultas distribuidas cuando:
- Necesitas combinar datos de múltiples fuentes
- Requieres transacciones que abarquen varios servidores
- Trabajas con sistemas heterogéneos

Conviene usar OPENQUERY cuando:
- El servidor remoto no es SQL Server
-Se desea aprovechar la potencia del servidor remoto
- Se necesita usar comandos específicos del sistema remoto

Es recomendable usar 4-Part Name cuando:
- Ambos servidores son SQL Server
- SE Prefiere sintaxis estándar de SQL
- SEe requiere un optimizador local gestione la consulta

DESARROLLO

Instrucciones: Considerar la base de datos de covidHistorico y proponer una fragmentación por regiones (recuperar una clasificación de regiones políticas y/o económicas del país) y alojar cada fragmento en una base de datos distinta. Para la distribución de los fragmentos considerar 4 nodos (al menos uno con un servidor MYSQL). Interconectar los nodos y modificar las consultas 3,4,5 y 7 de la practica 1 para que sean consultas distribuidas.

CONEXIÓN Y FRAGMENTACIÓN

Primero para conectar los nodos decidimos la siguiente clasificación de las regiones económicas del país

por lo que para cumplir con los 4 nodos de la practica se agruparon por esta clasificación

- Norte: incluyó Norte, Noreste y Noroeste
- Sur: incluyó Pacífico Sur, Península de Yucatán y Tabasco
- Oriente: incluyó Centro sur y Veracruz
- Occidente y otros: incluyó todo lo demás. (Aquí es donde se incluyen datos con regiones no consideradas)

Y en base a la lista de entidades de la base de datos es como al final en SQL Server se hizo la creación de los nodos

-- NORTE

ENTIDAD_RES IN ('08','05','10','32','24','25','26','18','02','03')

-- SUR

ENTIDAD_RES IN ('07','12','20','27','31','23','04')

-- ORIENTE
ENTIDAD_RES IN ('09','13','21','22','15','17','29','30')

-- OCCIDENTE Y OTROS
ENTIDAD_RES NOT IN ('07','12','20','27','31','23','04','08','05','10','32','24','25','26','18','02','03','09','13','21','22','15','17','29','30')

En base a lo anterior nuestra red finalmente se declara de la siguiente manera

Estructura de red

Para cada fragmento se usó un servidor, 3 SQL Servers y uno MySQL. Para la implementación se intentó usar directamente la conexión a MySQL desde los tres nodos extra, pero se descartó porque sería necesario configurar en cada nodo el ODBC para MySQL, que no es automatizable y requería privilegios de administrador los cuales no se tienen acceso en el laboratorio.

Entonces analizando la situación del laboratorio:

- El ODBC y Linked Server ya está preconfigurado en cada máquina.
- Las máquinas están en una red local aislada.
- No se tienen permisos de administrador en el laboratorio.

Optamos por usar un servidor SQL Server como proxy entre los nodos SQL Server y MySQL. Esta configuración permite reutilizar un nodo SQL Server, pero en esta implementación se usó un nodo extra que solo actua como proxy.

Fragmentación de datos

Existen varios caminos para llevar a cabo la fragmentación y distribución de datos a cada nodo. Para el caso específico de servidores SQL Server:

1. Fragmentar, respaldar/copiar en un archivo y restaurar en el servidor destino. (Esto es bastante bueno si el acceso a las máquinas físicas es rápido)
2. Copiar todos los datos de un servidor a otro y luego fragmentar en el servidor destino. (Esto es desperdicio de recursos, ya sea física o remota)
3. Fragmentar y luego insertar los datos de forma remota en el servidor destino. (Este es viable si no se tiene acceso físico a los servidores, pero está limitado por el ancho de banda y puede ser riesgoso/complejo en caso de fallo de red o de los servidores)

En el laboratorio pudimos haber usado la primera opción, pero decidimos usar la 3er opción por comodidad

Por otra parte, para el caso de MySQL hay más retos, ya que no hay una forma tan sencilla de copiar los datos de SQL Server a MySQL. Y cualquier persona sin experiencia se topará con procesos lentos, complejos y con riesgos de fallos. Las opciones son:

1. Fragmentar, exportar los datos a CSV y restaurar en MySQL. Remoto o local.
2. Fragmentar e insertar de forma remota.

3. Exportar los datos a CSV, restaurar en MySQL y luego fragmentar.

Por la Estructura de Red se descartó la segunda opción ya que era complejo hacer la inserción desde el proxy (este no tiene los datos). La primera opción en su versión remota podría ser útil en un caso donde se tenga más libertad de permisos, pero al estar en el laboratorio la descarté, además de que estaría limitado por el ancho de banda. En cuanto a la versión local, la intenté, pero tuve algún error, probablemente debido a la falta de experiencia, y preferí usar la tercera opción.

Para la tercera opción también hubo problemas:

- Restaurarlo desde el Wizard de Workbench de MySQL era extremadamente lento, insertaba línea por línea.
- Restaurarlo por comando LOAD desde el Workbench de MySQL era un poco menos lento, pero trababa la aplicación y al final perdía la conexión, lo cual generó varios problemas de filas duplicadas.

Al final usamos el comando LOAD desde la línea de comandos de MySQL, que es más rápido y no pierde la conexión. El tiempo estimando para restaurar los datos fue de 3 minutos. (Se tiene que habilitar el comando LOAD en la configuración de MySQL, https://stackoverflow.com/questions/10762239/mysql-enable-load-data-local-infile)

Distribución de las consultas

En primer instancia, fue intuitivo cambiar el origen de datos de las consultas ya desarrolladas a la unión de los datos de todos los nodos y esto funcionaba, sin embargo, era lento porque transmitía todas las filas de cada nodo al nodo que ejecutaba la consulta, lo cual no era distribuir la consulta sino hacerla remota. Una vez observado eso empecé a usar OPENQUERY para que los nodos hicieran un preprocesado de los datos para minimizar el número de filas a transmitir.

Al empezar a hacer el cambio fue donde salieron los detalles de cada consulta. Por ejemplo, en la consulta 3, se requiere hacer un conteo en cada nodo para obtener el total de casos por enfermedad y de los registrados, para después en el nodo inicial hacer el cálculo final uniendo los resultados de todos los nodos. También fue un problema que en la consulta original se usaba IIF que no es estándar y no existe en MySQL, por lo que se tuvo que adaptar la consulta para que funcione en ambos.

Otro problema principal es que los tipos de datos en los motores de SQL no fueron fáciles de adaptar, por lo cual algunas columnas quedaron con tipos de datos incorrectos. Ej: las fechas se convirtieron a text, algunos que pudieron ser char se quedaron como text. Por esto surgieron otros problemas de tipo de datos que se solventaron con CAST.

Proxy

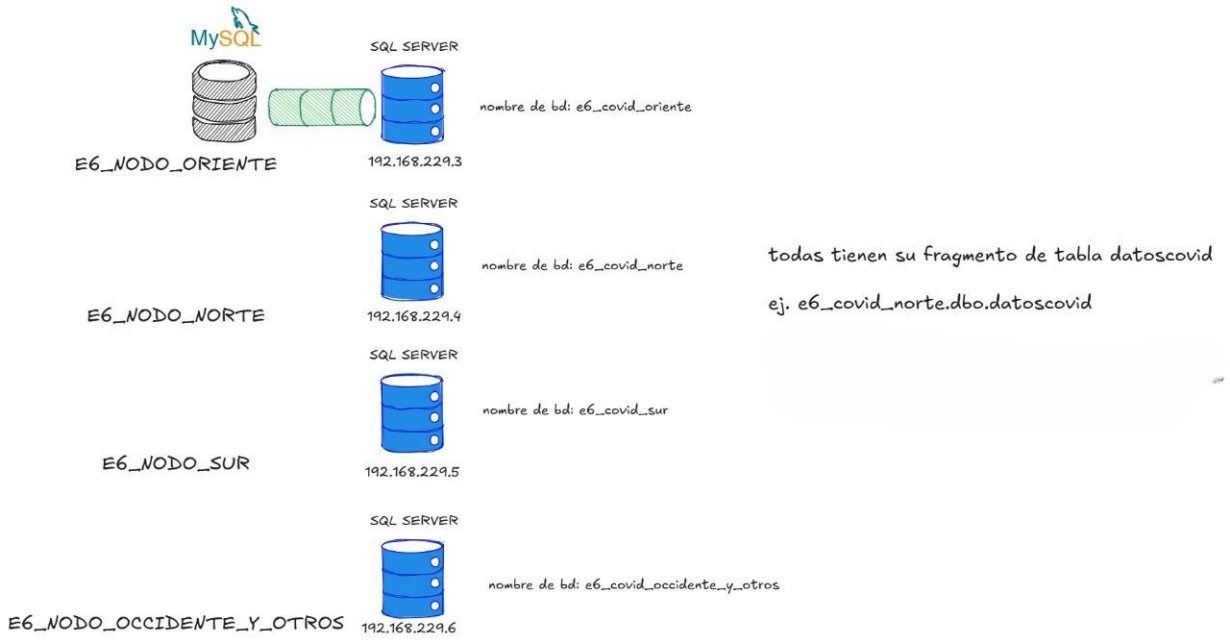
Es de resaltar la forma en la que funciona el proxy, ya que en primer lugar pensé usar una vista para hacer la consulta y la descarté porque creí que esto causaría el mismo problema de consulta remota y no distribuida, en realidad no sé si hay alguna forma de hacerlo. Pudo haber sido un procedimiento almacenado, pero decidí mantenerlo simple y anidar dos OPENQUERY.

Para hacer la anidación se requiere que los Linked Servers tengan configurado el RPC y autenticación propia.

Ya configurado se puede anidar tal que así:

```
SELECT *
FROM
    OPENQUERY(E6_NODO_ORIENTE,
    'SELECT *
     FROM OPENQUERY(MYSQL_8,
     ''
     SELECT
        SUM(CASE WHEN DIABETES = 1 THEN 1 ELSE 0 END) AS CASOS_DIABETES,
        SUM(CASE WHEN HIPERTENSION = 1 THEN 1 ELSE 0 END) AS CASOS_HIPERTENSION,
        SUM(CASE WHEN OBESIDAD = 1 THEN 1 ELSE 0 END) AS CASOS_OBESIDAD,
        COUNT(*) AS CASOS_TOTALES,
        ''''ORIENTE'''' AS REGION
     FROM e6_covid_oriente.datoscovid
     WHERE CLASIFICACION_FINAL IN (1, 2, 3)
     '')
')
```

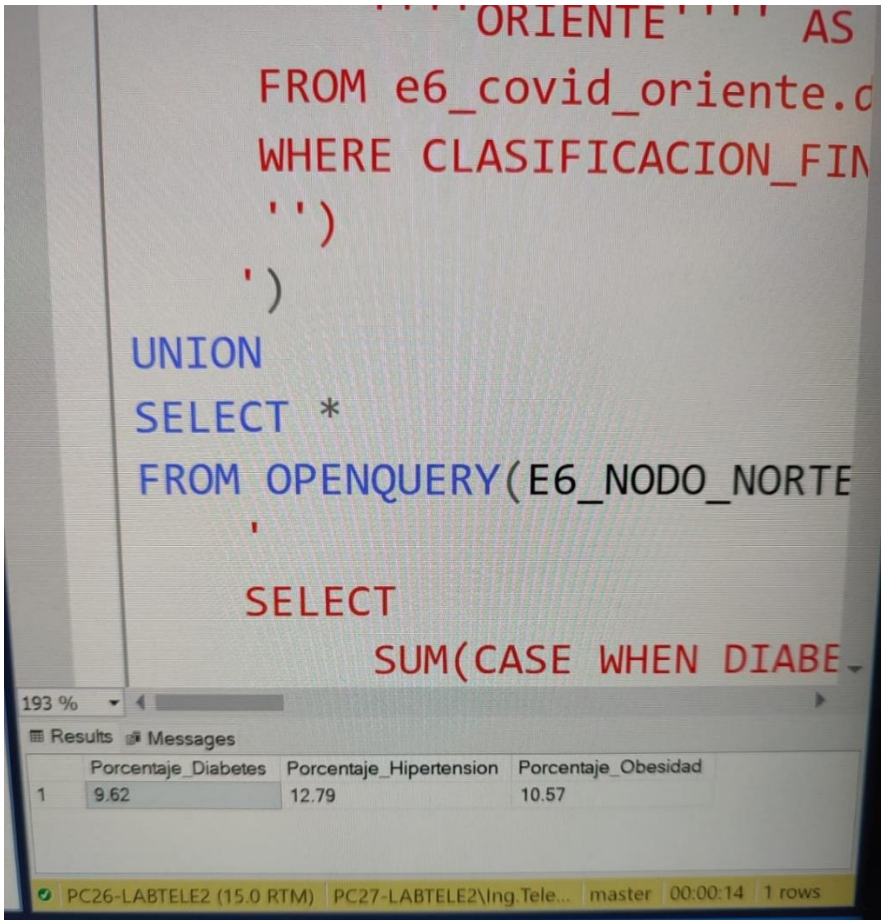
En el siguiente esquemático se muestran las maquina del laboratorio que utilizamos para hacer la práctica y que nodo es cada una



CONSULTAS

Consulta 3

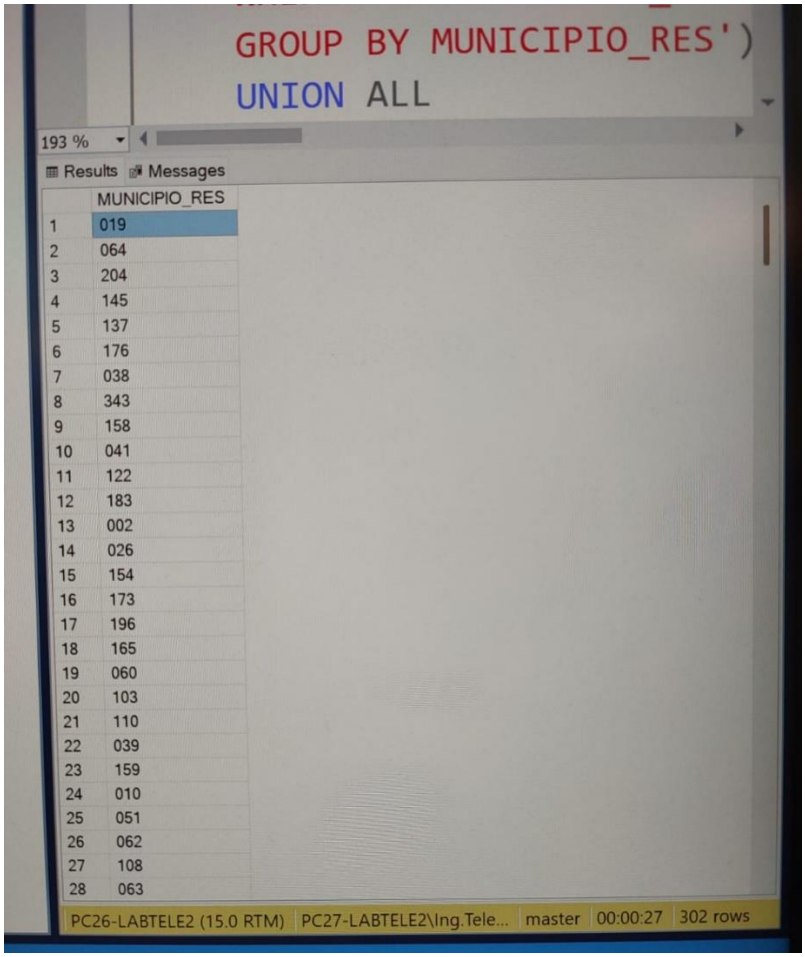
Consulta 03	Listar el porcentaje de casos confirmados en cada una de las siguientes morbilidades a nivel nacional: diabetes, obesidad e hipertensión.
Requisitos	N/A
Responsable	Keb Sebastián Fernández Guerrero
Catálogos	OBESIDAD, HIPERTENSION y DIABETES son catálogos SI/NO, donde 1 es SI.
Comentarios	**Se usa IIF en lugar de CASE por simplificar la sintaxis**



Respecto a la consulta original no cambia mucho, simplemente la suma se hace por nodo y al final el nodo invocador hace el cálculo final.

Consulta 4

Consulta 04	Listar los municipios que no tengan casos confirmados en todas las morbilidades: hipertensión, obesidad, diabetes y tabaquismo.
Requisitos	- Mostrar el nombre del municipio. - Mostrar solo los municipios que tienen casos no confirmados en todas de las morbilidades.
Responsable	Armando Eduardo Sánchez Herrera
Catálogos	CLASIFICACION_FINAL: 1, 2, 3 = Casos confirmados. - HIPERTENSION, OBESIDAD, DIABETES, TABAQUISMO: 1 = Sí, 0 = No.
Comentarios	- Se utiliza una CTE para calcular el número de casos confirmados para cada morbilidad por municipio. - Se filtran los municipios que no tienen casos confirmados en las entidades



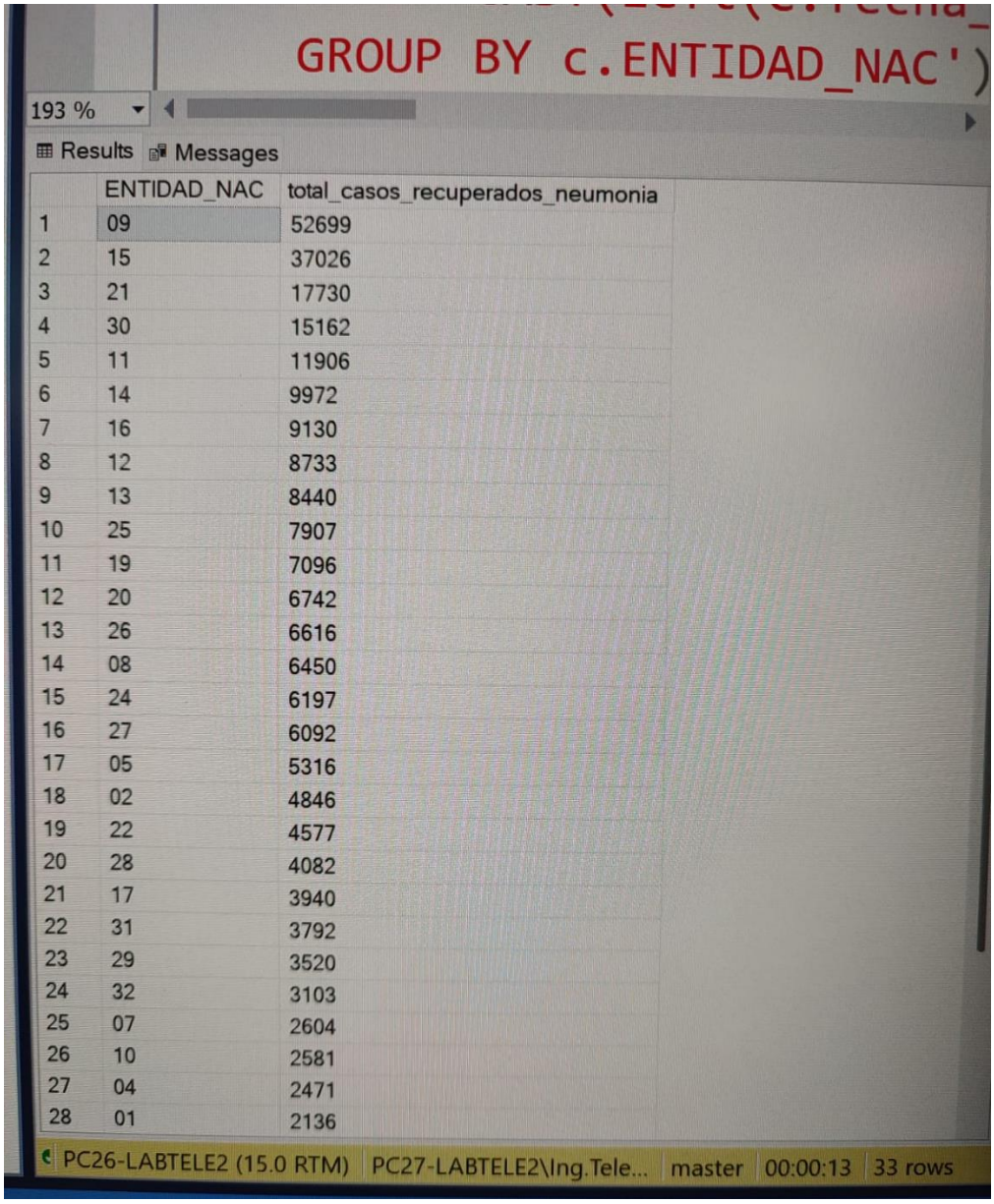
Más allá del CAST en la columna de MySQL, acá hay un detalle importante en los GROUP BY, anteriormente se usaba un WHERE final validando los casos directamente:

```
WHERE casos_hipertension > 0
  AND casos_obesidad > 0
  AND casos_diabetes > 0
  AND casos_tabaquismo > 0
GROUP BY MUNICIPIO_RES;
```

Sin embargo al hacer la fragmentación algunos municipios se encuentran en varios nodos (no sabemos por qué, simplemente los datos vienen así) y por ende los registros no quedaban compactados correctamente en los resultados de las consultas distribuidas, esto hacía que se descartaran ciertos datos y el resultado fuera erróneo, entonces decidimos modificarlo para que se agruparan todos los datos sin importar los casos y posteriormente filtrarlos con el HAVING.

```
GROUP BY MUNICIPIO_RES
HAVING SUM(casos_hipertension) > 0
  AND SUM(casos_obesidad) > 0
  AND SUM(casos_diabetes) > 0
  AND SUM(casos_tabaquismo) > 0;
```

Consulta 05	Listar los estados con más casos recuperados con neumonía.
Requisitos	- Buscar a los estados con esos casos recuperados -
Responsable	Juan Carlos Ramírez Orozco
Catálogos	- Uso la vista creada de casos confirmados que agrupa lo datos de todos aquellos usuarios con clasificación final 1,2,3
Comentarios	- Hacemos una consulta en la cual agrupemos los datos por estado y año. - De esa búsqueda buscamos el valor máximo para el cual haremos comparaciones con cada dato. - Hacemos una auto consulta donde buscamos eliminar todos los demás registros que no necesitamos tomar en cuenta. - Ordenamos y presentamos



	ENTIDAD_NAC	total_casos_recuperados_neumonia
1	09	52699
2	15	37026
3	21	17730
4	30	15162
5	11	11906
6	14	9972
7	16	9130
8	12	8733
9	13	8440
10	25	7907
11	19	7096
12	20	6742
13	26	6616
14	08	6450
15	24	6197
16	27	6092
17	05	5316
18	02	4846
19	22	4577
20	28	4082
21	17	3940
22	31	3792
23	29	3520
24	32	3103
25	07	2604
26	10	2581
27	04	2471
28	01	2136

El detalle más notable de esta consulta es el CAST de la fecha (el cual ya se hacía antes), solo que en MySQL INT es SIGNED.

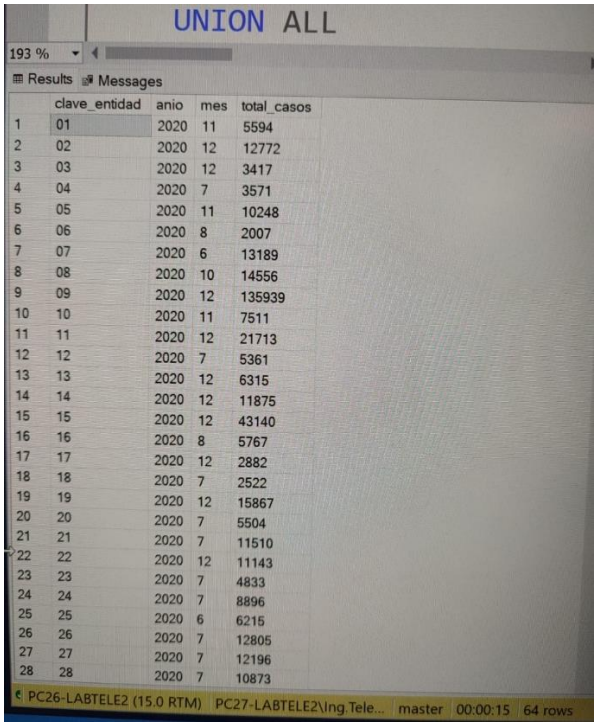
```
CAST(left(C.fecha_def,4) as signed) = 9999
```

Más allá de eso está el CAST para ENTIDAD_NAC en MySQL que es tipo TEXT originalmente, pero se convirtió a CHAR(2) para que funcionen las operaciones de SQL.

```
CAST(c.ENTIDAD_NAC as CHAR(2))
```

Consulta 7

Consulta 07	7. Para el año 2020 y 2021, cuál fue el mes con más casos registrados, confirmados y sospechosos, por estado registrado en la base de datos.
Requisitos	- Mostrar el nombre de la entidad. - Mostrar el mes con más casos por estado y año.
Responsable	Armando Eduardo Sánchez Herrera
Catálogos	- CLASIFICACION_FINAL: 1, 2, 3 = Casos confirmados; 6 = Casos sospechosos.
Comentarios	- Se utiliza una CTE para calcular el número de casos confirmados y sospechosos por estado, año y mes. - Se hace un JOIN con la tabla de catálogo de entidades para obtener el nombre de la entidad. - Se utiliza ROW_NUMBER() para asignar un ranking a cada mes dentro de cada estado y año. - Se filtran solo los meses con más casos por estado y año.



Acá el detalle es el CAST (en MySQL) de la fecha que es TEXT, pero lo pasamos como DECIMAL(4,0) para que funcionen las operaciones de SQL.

```
CAST(c.ENTIDAD_RES as CHAR(2)),
CAST(YEAR(c.fecha_ingreso) as decimal(4,0)) AS anio,
CAST(MONTH(c.fecha_ingreso) as decimal(4,0)) AS mes,
COUNT(*) AS total_casos
```

Nota: el "anio" es porque teníamos duda que causara problema, pero creo que podría ser "año" sin problema. De igual forma creo que el mes podría ser DECIMAL(2,0).

Conclusiones

En esta práctica reforzamos los últimos temas vistos, como el de servidores vinculados que nos permitió ejecutar las consultas desde cualquiera de las 3 maquinas que ocupamos, la partición que hicimos de servidores por regiones fue lo mejor que pudimos haber hecho pues cuando realizamos la practica 1 el tiempo de ejecución era elevado por la cantidad de registros que teníamos, esta vez tardamos menos, pero si se nos complicó el configurar correctamente los servidores eso fue lo que más tiempo nos llevó, además de que al usar MySQL la tabla ENTIDAD_RES era del tipo TEXT por lo que se tuvo que hacer un cast porque de lo contrario no se podía ejecutar el query, otro reto fue pensar bien que estados serian en cada región pero al final se opto por tomar como referencia los puntos cardinales, lo cual ayudo bastante pues consideramos que los fragmentos quedaron balanceados. Esta fue una buena práctica, pero si tuvimos esos problemas lo cual hizo que nos tardáramos mas de dos semanas en completarla, uno de los mayores retos que enfrentamos fue la interoperabilidad entre diferentes gestores de bases de datos, dió problemas en todas las etapas, fragmentación y distribución de las consultas. También

respecto al rendimiento notamos que pese a que se supone las consultas distribuidas pueden acelerar el cómputo un poco, no se apreció en esta práctica, en parte porque MySQL fue un gran cuello de botella, aunque no sabemos la razón exacta podría ser una mala configuración, pero también podría ser inherente al gestor de BD, hay otras razones, tal vez por la cantidad de datos o tal vez porque los servidores no están muy cargados.

Esta práctica sirvió como un excelente ejercicio para comprender la brecha entre la teoría de bases de datos distribuidas y su implementación práctica. Los desafíos encontrados - desde problemas de interoperabilidad hasta cuellos de botella inesperados - proporcionaron aprendizajes valiosos que difícilmente se obtienen solo con estudio teórico.

El mayor aprendizaje fue que el diseño de sistemas distribuidos requiere un equilibrio cuidadoso entre múltiples factores: distribución de datos, consistencia, rendimiento y mantenibilidad. Aunque no se lograron las mejoras de rendimiento esperadas, la experiencia obtenida en resolver problemas reales de integración entre sistemas heterogéneos es invaluable para nuestro desarrollo como ingenieros de bases de datos.

Queda claro que en entornos de producción sería esencial realizar pruebas más exhaustivas de configuración y rendimiento, así como contar con mayor control sobre los entornos de ejecución. Esta práctica sienta las bases para abordar proyectos más ambiciosos de distribución de datos en el futuro.