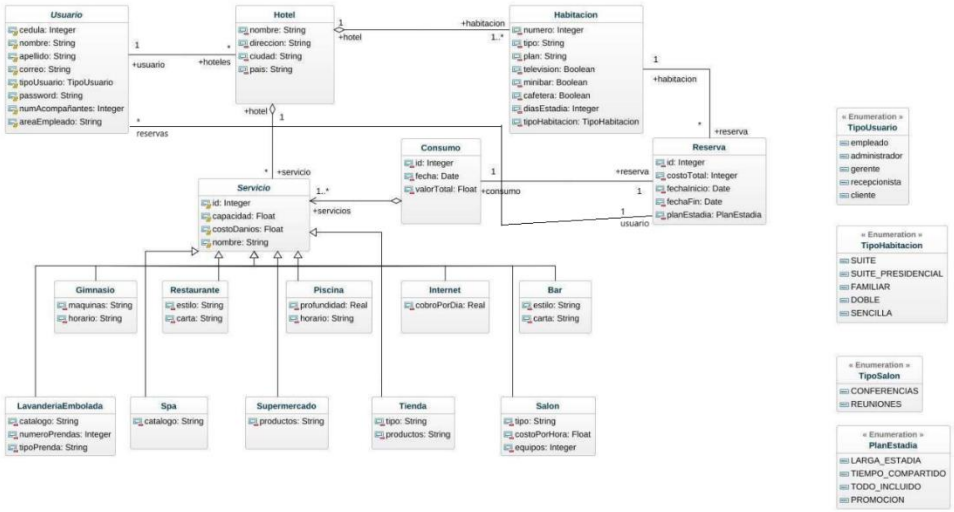
Iteración 2
Sistemas Transaccionales

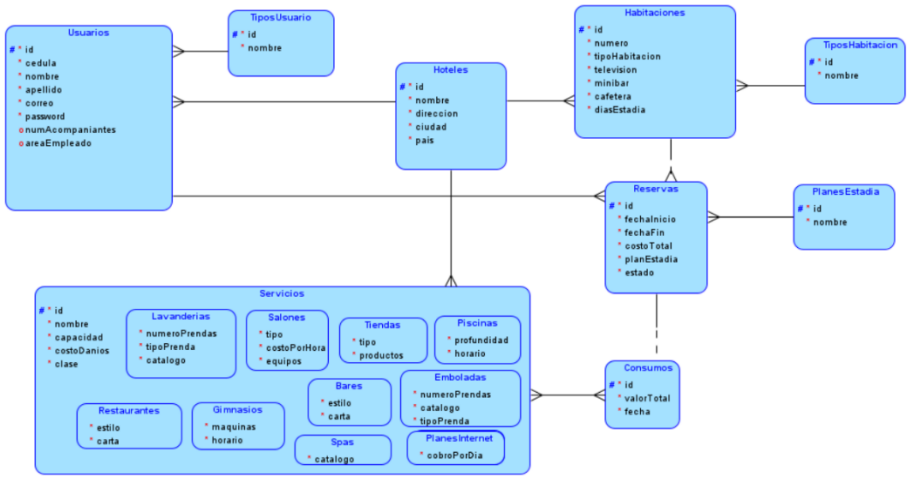
Ajuste al UML

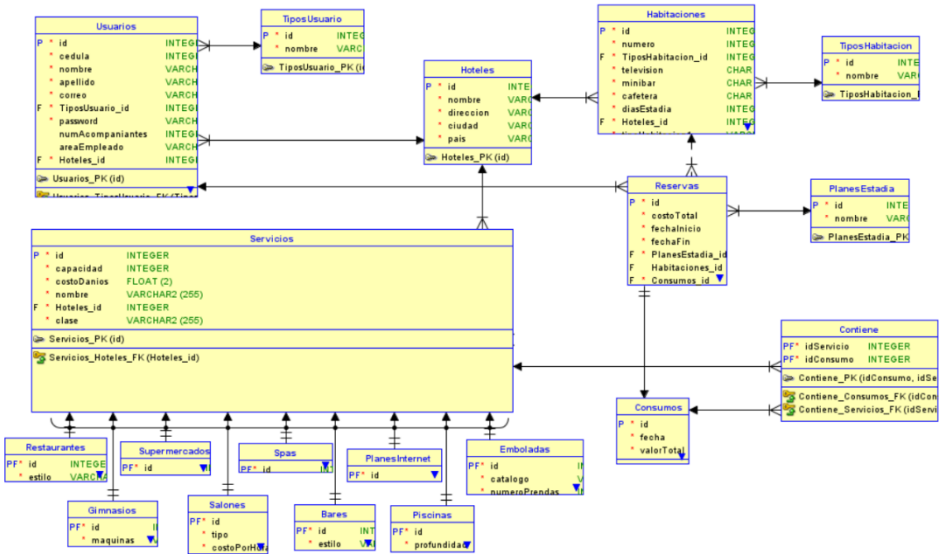
En la fase de revisión y optimización de nuestro proyecto, identificamos la necesidad de refinar nuestro esquema UML para mejorar la eficiencia operativa del sistema. La entrega anterior reveló que algunos de los requerimientos funcionales demandaban una relación más directa y eficiente entre las entidades de reservas y usuarios. Ante esta situación, decidimos implementar un cambio estructural en nuestro modelo de datos.

La modificación consistió en establecer una relación uno a muchos (1:N) directamente entre usuarios y reservas. Con este cambio, cada usuario en el sistema puede estar asociado con múltiples reservas, lo que refleja con mayor precisión la dinámica real de las operaciones hoteleras, donde un cliente puede realizar varias reservas a lo largo del tiempo.Así mismo, se cambió la relación Usuario-Hotel por una ManyToOne para que un usuario pertenezca sólo a un hotel. Esto elimina la necesidad de la tabla Tiene originalmente planteada. Además, se agregó un atributo “estado” a la Reserva para poder modelar las operaciones de Check-In y Check-Out.



Este ajuste en el esquema UML tiene como objetivo principal simplificar las consultas y operaciones en la base de datos. Al reducir la necesidad de múltiples joins entre tablas para obtener la información relacionada con las reservas de un usuario, mejoramos significativamente la eficiencia de las consultas. Esto no solo optimiza el tiempo de respuesta del sistema sino que también facilita la comprensión y mantenimiento del código, ya que las relaciones entre las entidades son más intuitivas y directas.





De esta manera, se alteró el diagrama de Entidad-Relación y el diagrama Relacional.

Diseño de consultas.

1. Requerimiento Funcional 1

   Para presentar el dinero recolectado por los servicios en cada habitación, se desarrolló una consulta SQL específica. Inicialmente, esta consulta se ejecutaba en 0.3 segundos con un costo de ejecución de 145. Al analizar el plan de ejecución, observamos que la consulta realizaba un recorrido completo de las tablas de consumos y reservas para aplicar el filtro del último año, tal como se ilustra en la Figura 1.1. Se constató que Oracle no había generado automáticamente un índice para optimizar este proceso.

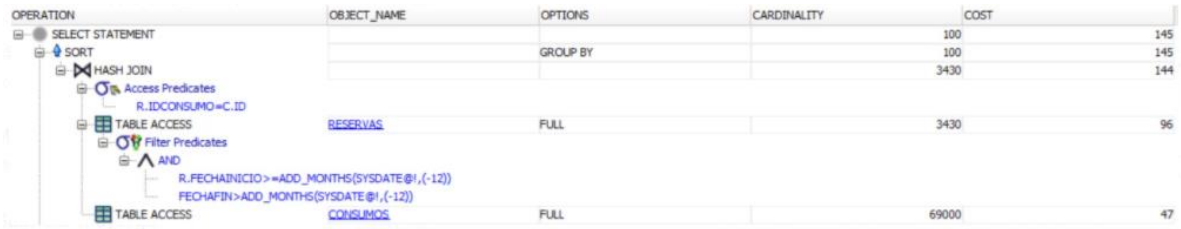


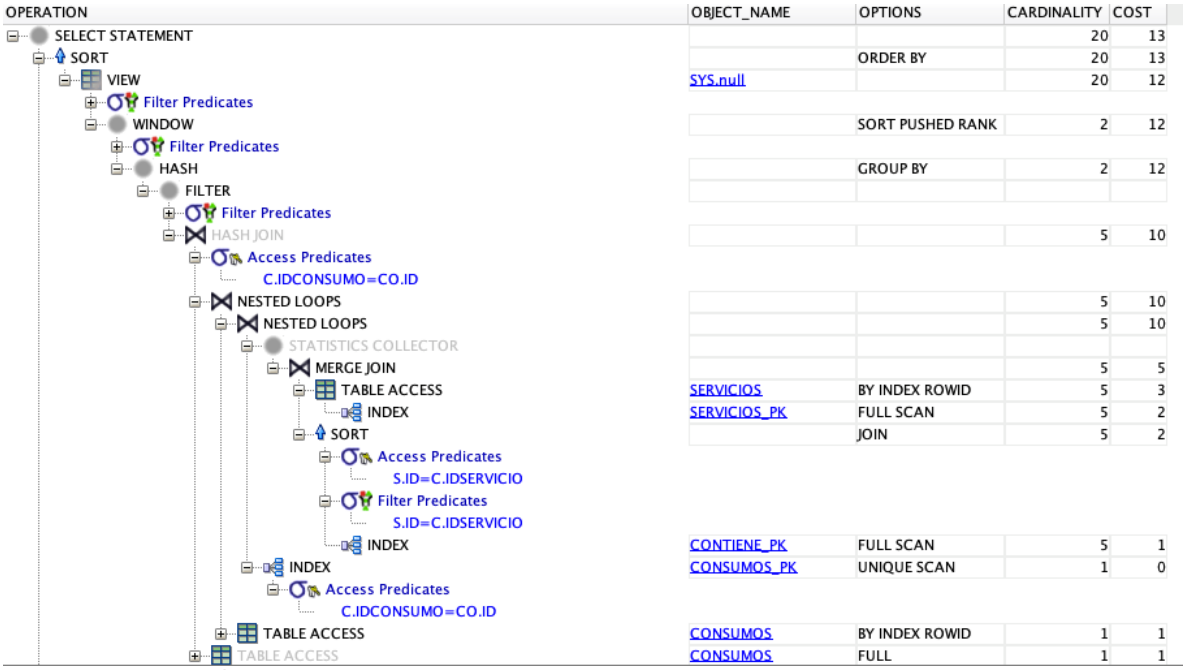
Figura 1.1 Plan de ejecución de la consulta 1 generada por Oracle.

A pesar de la posibilidad de optimización, determinamos que la implementación de índices para este requerimiento funcional no era imprescindible. La consulta ya demostraba un rendimiento eficiente, con un tiempo promedio de búsqueda de 0.03 segundos para recuperar los datos de las 100 habitaciones. En este contexto, decidimos no implementar índices adicionales para evitar impactar negativamente en la velocidad de las operaciones de inserción, actualización o eliminación, las cuales no justifican la necesidad de optimización dada la rapidez actual de la consulta.

2. Requerimiento Funcional 2

Para el requerimiento funcional que busca destacar los 20 servicios más solicitados, se optó por no añadir índices adicionales. Esta decisión se basó en el rendimiento ya óptimo de la consulta SQL, que se ejecutó en 0.067 segundos con un costo computacional de 13, incluso frente a un volumen de datos considerable. Por lo tanto, se consideró innecesario el uso de índices adicionales que no fueran estrictamente necesarios.

Oracle, por su parte, implementó dos índices de manera autónoma. El primero es un índice primario en la tabla de servicios, asociado a la clave primaria, lo cual facilita la operación de merge join con la tabla de consumos. El segundo índice primario se estableció en la tabla de consumos, lo cual es particularmente beneficioso ya que permite realizar una búsqueda única (unique scan) durante las operaciones de bucles anidados (nested loops) entre las tablas de 'contiene' y 'consumos'. Estos índices generados automáticamente por Oracle contribuyen a la eficiencia de las consultas sin que sea necesario intervenir manualmente en la estructura de índices de la base de datos.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			20	13
SORT		ORDER BY	20	13
VIEW	SYS.null		20	12
Filter Predicates				
WINDOW		SORT PUSHED RANK	2	12
Filter Predicates				
HASH		GROUP BY	2	12
FILTER				
Filter Predicates				
HASH JOIN			5	10
Access Predicates				
C.IDCONSUMO=CO.ID				
NESTED LOOPS			5	10
NESTED LOOPS			5	10
STATISTICS COLLECTOR				
MERGE JOIN			5	5
TABLE ACCESS	SERVICIOS	BY INDEX ROWID	5	3
INDEX	SERVICIOS_PK	FULL SCAN	5	2
SORT		JOIN	5	2
Access Predicates				
S.ID=C.IDSERVICIO				
Filter Predicates				
S.ID=C.IDSERVICIO				
INDEX	CONTIENE_PK	FULL SCAN	5	1
INDEX	CONSUMOS_PK	UNIQUE SCAN	1	0
Access Predicates				
C.IDCONSUMO=CO.ID				
TABLE ACCESS	CONSUMOS	BY INDEX ROWID	1	1
TABLE ACCESS	CONSUMOS	FULL	1	1

Figura 2.1 Plan de ejecución de la consulta 2 generada por Oracle.

3. Requerimiento Funcional 3

Para el requerimiento de calcular el índice de ocupación por habitación, tanto Oracle como nuestro equipo decidimos no implementar índices adicionales. La consulta actual se ejecuta en un tiempo muy eficiente de 0.045 segundos y presenta un costo computacional de 98, lo cual indica un rendimiento satisfactorio. El plan de ejecución solo lee una única vez la tabla reservas por completo para hacer las operaciones de filtrado.

Pese a esta lectura y dado que la velocidad de ejecución es ya bastante rápida y el costo es bajo, la introducción de índices adicionales podría no justificarse, especialmente si consideramos el impacto potencial en las operaciones de escritura como inserciones, actualizaciones y eliminaciones. Mantener la base de datos sin índices adicionales en este escenario específico permite equilibrar el rendimiento de lectura con la eficiencia de las operaciones de escritura, asegurando así un manejo óptimo de los recursos del sistema.

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			100	98
HASH		GROUP BY	100	98
TABLE ACCESS	RESERVAS	FULL	3439	97
Filter Predicates				
AND				
FECHAINICIO >=ADD_MONTHS(TRUNC(SYSDATE@!),(-12))				
FECHAFIN >ADD_MONTHS(TRUNC(SYSDATE@!),(-12))				
Other XML				

Figura 3.1 Plan de ejecución de la consulta 3 generada por Oracle.

4. Requerimiento Funcional 4

La ejecución de la consulta SQL para mostrar los servicios que cumplen con ciertos criterios se completó de manera eficiente, con un tiempo de respuesta de 0.042 segundos y un costo computacional de 152 unidades. Oracle ha optimizado automáticamente esta consulta mediante la creación de índices que mejoran significativamente el rendimiento.

El primer índice se aplicó durante la operación de hash join entre las tablas contiene y consumos. Este índice ha demostrado ser particularmente efectivo, permitiendo realizar un "fast full scan". Esto implica que Oracle puede escanear rápidamente los índices en lugar de las tablas completas, lo que reduce el tiempo de acceso a los datos necesarios para la consulta.

El segundo índice fue creado por Oracle en la tabla reservas para facilitar su integración con las demás tablas a través de un hash join. Este índice permite realizar un "unique scan", lo que significa que Oracle puede buscar directamente entradas únicas en el índice, lo que resulta en una búsqueda más rápida y un costo muy bajo de solo 1 unidad.

Estos índices automáticos son suficientes para mantener un rendimiento óptimo de la consulta. Por lo tanto, hemos decidido no agregar índices adicionales manualmente. Esto nos permite mantener un equilibrio adecuado entre el rendimiento de las consultas de lectura y la eficiencia de las operaciones de escritura, como insertar, actualizar o eliminar registros, que pueden verse afectadas negativamente por la sobrecarga de índices adicionales.

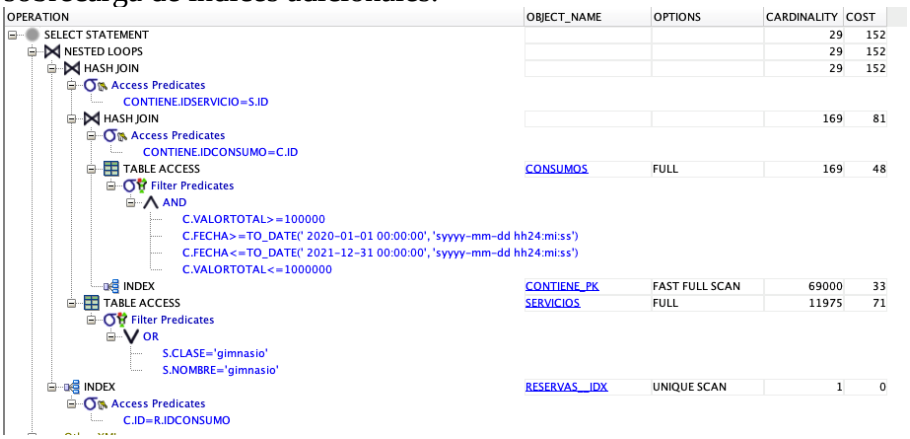


Figura 4.1 Plan de ejecución de la consulta 4 generada por Oracle.

5. Requerimiento Funcional 5

Para mejorar el rendimiento de la consulta que muestra el consumo total de un usuario específico dentro de un rango de fechas, se implementó un índice secundario no único en la columna idusuario de la tabla reservas. La consulta original se ejecutaba en 0,03 segundos con un costo de ejecución de 98. Oracle había creado automáticamente dos índices: un índice único en la tabla usuarios para optimizar la búsqueda por id de

usuario, y un índice en la tabla consumos para mejorar la eficiencia del bucle anidado con la tabla reservas.

Sin embargo, el análisis del plan de ejecución reveló que el principal cuello de botella era el acceso completo a la tabla reservas. Para resolver este problema, creamos un índice secundario simple con el comando CREATE INDEX idx_reservas_idusuario ON reservas(idusuario). Este índice no es único, ya que múltiples reservas pueden estar asociadas a un solo usuario. La adición de este índice redujo significativamente el costo de la consulta de 98 a 3, lo que indica una mejora sustancial en la eficiencia. Con este índice, la base de datos ya no necesita realizar un escaneo completo de la tabla reservas; en cambio, puede acceder directamente a las filas relevantes utilizando el idusuario, lo que resulta en una operación mucho más rápida.

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	98
HASH		GROUP BY	1	98
NESTED LOOPS			1	98
NESTED LOOPS			1	98
NESTED LOOPS			1	97
TABLE ACCESS	USUARIOS	BY INDEX ROWID	1	2
INDEX	USUARIOS_PK	UNIQUE SCAN	1	1
Access Predicates				
U.ID=1				
TABLE ACCESS	RESERVAS	FULL	1	95
Filter Predicates				
R.IDUSUARIO=1				
INDEX	CONSUMOS_PK	UNIQUE SCAN	1	0
Access Predicates				
R.IDCONSUMO=C.ID				
TABLE ACCESS	CONSUMOS	BY INDEX ROWID	1	1
Filter Predicates				
AND				
C.FECHA>=TO_DATE(' 2022-12-11 00:00:00', 'syyyy-mm-dd hh24:mi:ss')				
C.FECHA<=TO_DATE(' 2023-12-12 00:00:00', 'syyyy-mm-dd hh24:mi:ss')				

Figura 5.1 Plan de ejecución de la consulta 5 generada por Oracle.

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	3
HASH		GROUP BY	1	3
NESTED LOOPS			1	3
NESTED LOOPS			1	3
NESTED LOOPS			1	2
TABLE ACCESS	USUARIOS	BY INDEX ROWID	1	2
INDEX	USUARIOS_PK	UNIQUE SCAN	1	1
Access Predicates				
U.ID=1				
TABLE ACCESS	RESERVAS	BY INDEX ROWID B...	1	0
INDEX	IDX_RESERVAS_IDUSU...	RANGE SCAN	1	0
Access Predicates				
R.IDUSUARIO=1				
INDEX	CONSUMOS_PK	UNIQUE SCAN	1	0
Access Predicates				
R.IDCONSUMO=C.ID				
TABLE ACCESS	CONSUMOS	BY INDEX ROWID	1	1
Filter Predicates				
AND				

Figura 5.2 Plan de ejecución de la consulta 5 generada por Oracle con nuestros indices.

6. Requerimiento funcional 6

El requerimiento funcional 6 se desglosa en tres consultas específicas relacionadas con las fechas y la ocupación de habitaciones, así como los ingresos generados.

- Fecha con Mayor Ocupación de Habitaciones: La consulta para identificar la fecha con la mayor ocupación de habitaciones se ejecuta en 0,089 segundos y

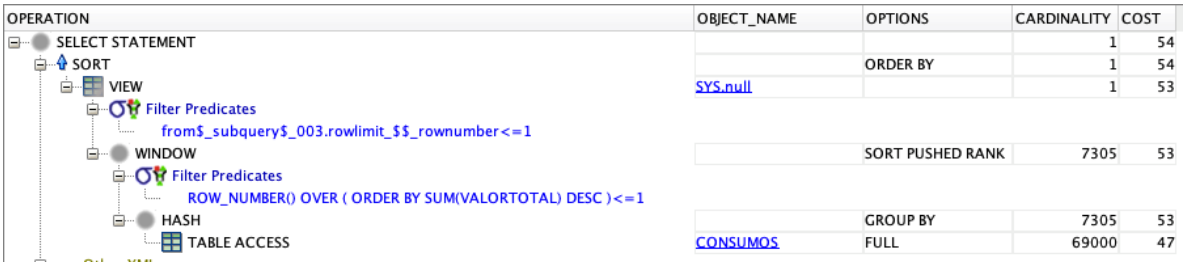
tiene un costo computacional de 101. A pesar de que el principal factor de costo es el escaneo completo de la tabla de reservas, no se ha considerado la implementación de un índice. La razón es que la presencia de valores nulos en los datos podría disminuir la efectividad de cualquier índice creado, ya que los índices tienden a ser menos eficientes en la presencia de numerosos valores nulos. Por lo tanto, hemos decidido no implementar un índice adicional, como se ilustra en la Figura 6.1 del informe.

- Fecha con Mayores Ingresos: La consulta que busca la fecha con los ingresos más altos tarda 0,185 segundos en ejecutarse y tiene un costo de 54. Aunque se realiza un escaneo completo de la tabla de consumos, no se ha generado automáticamente un índice, ni se ha considerado necesario crear uno manualmente. Esto se debe a que la tabla maneja una cantidad significativa de valores nulos, y la creación de un índice no se traduciría en una mejora sustancial del rendimiento de la consulta. Esta decisión se detalla en la Figura 6.2.

- Fecha con Menor Demanda: La última consulta, que busca la fecha con la menor demanda de habitaciones, se procesa en 0,085 segundos y tiene un costo de 101. Por razones similares a la primera consulta, no se ha optado por la creación de índices. La presencia de valores nulos y el rendimiento ya aceptable de la consulta sin índices adicionales justifican esta elección. Los detalles se pueden encontrar en la Figura 6.3.

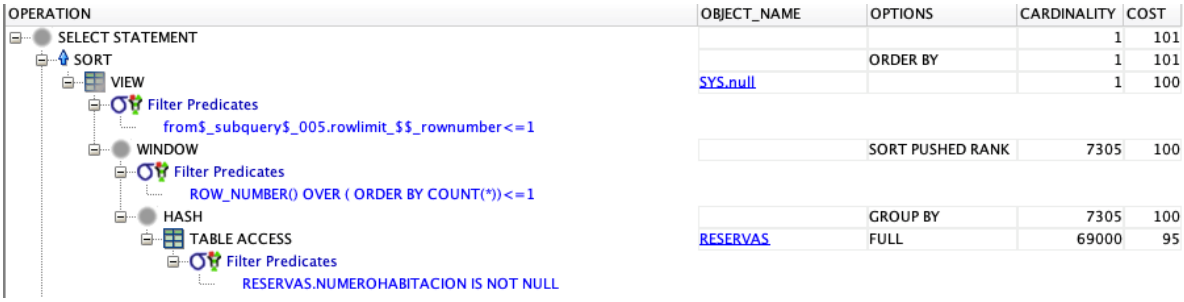
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	101
SORT		ORDER BY	1	101
VIEW	SYS.null		1	100
Filter Predicates				
from$_subquery$_005.rowlimit_$$_rownumber<=1				
WINDOW		SORT PUSHED RANK	7305	100
Filter Predicates				
ROW_NUMBER() OVER (ORDER BY COUNT(*) DESC)<=1				
HASH		GROUP BY	7305	100
TABLE ACCESS	RESERVAS	FULL	69000	95
Filter Predicates				
RESERVAS.NUMEROHABITACION IS NOT NULL				

Figura 6.1 Plan de ejecución de la consulta 6a generada por Oracle.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	54
SORT		ORDER BY	1	54
VIEW	SYS.null		1	53
Filter Predicates				
from$_subquery$_003.rowlimit_$$_rownumber<=1				
WINDOW		SORT PUSHED RANK	7305	53
Filter Predicates				
ROW_NUMBER() OVER (ORDER BY SUM(VALORTOTAL) DESC)<=1				
HASH		GROUP BY	7305	53
TABLE ACCESS	CONSUMOS	FULL	69000	47

Figura 6.2 Plan de ejecución de la consulta 6b generada por Oracle.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	101
SORT		ORDER BY	1	101
VIEW	SYS.null		1	100
Filter Predicates				
from$_subquery$_005.rowlimit_$$_rownumber<=1				
WINDOW		SORT PUSHED RANK	7305	100
Filter Predicates				
ROW_NUMBER() OVER (ORDER BY COUNT(*))<=1				
HASH		GROUP BY	7305	100
TABLE ACCESS	RESERVAS	FULL	69000	95
Filter Predicates				
RESERVAS.NUMEROHABITACION IS NOT NULL				

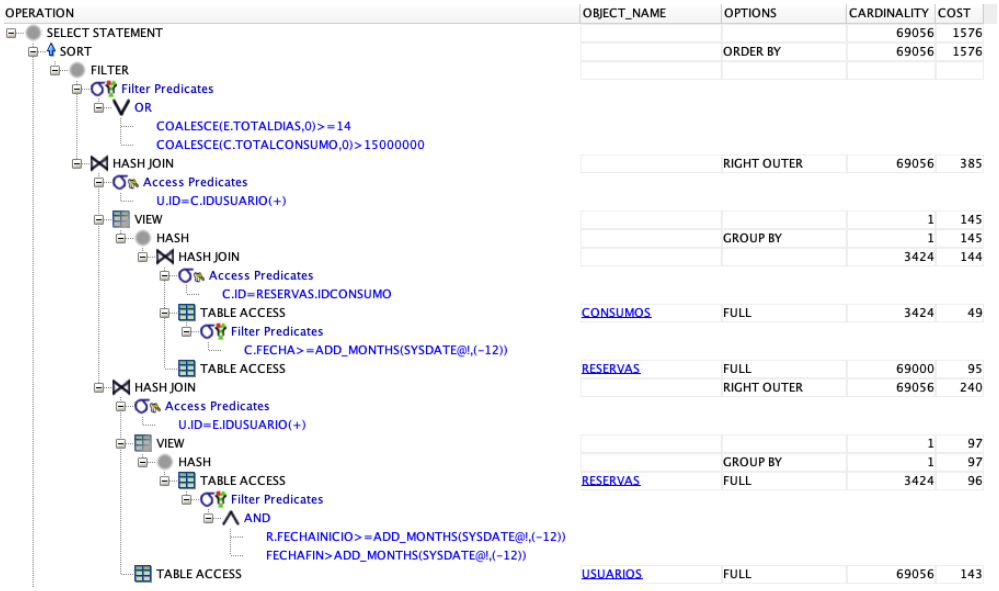
Figura 6.3 Plan de ejecución de la consulta 6c generada por Oracle.

7. Requerimiento funcional 7

Para mejorar la eficiencia de la consulta SQL destinada a identificar a los clientes más valiosos, se realizó un análisis exhaustivo del plan de ejecución. Originalmente, la consulta tardaba 2,5 segundos en ejecutarse y tenía un costo computacional de 1576 unidades. Al examinar el plan de ejecución, se observó que Oracle no había creado índices automáticos que pudieran optimizar la consulta. Este comportamiento se atribuye a la presencia de valores nulos en ciertas columnas críticas, como valortotal en la tabla consumos, lo cual es una situación común cuando no todos los clientes realizan consumos.

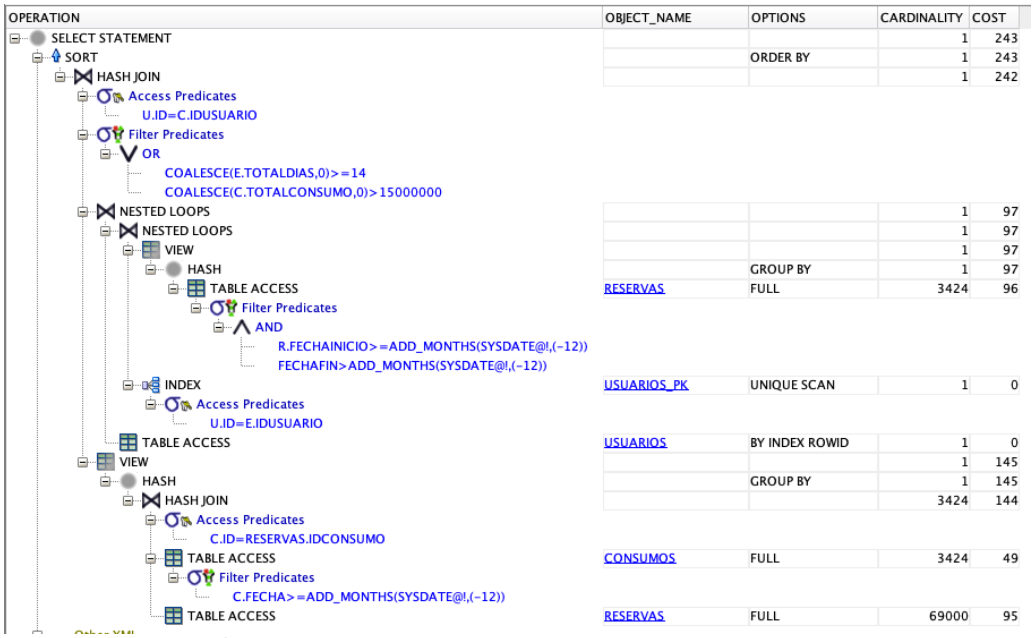
El análisis detallado reveló que los LEFT JOIN utilizados en la consulta eran particularmente costosos, representando más de 1000 unidades del costo total. Esto se debía a que estos LEFT JOIN estaban diseñados para manejar posibles valores nulos, pero, dada la estructura de la base de datos, se determinó que podían ser reemplazados por NATURAL JOIN sin perder funcionalidad y garantizando la integridad de los datos.

Al reemplazar los LEFT JOIN por NATURAL JOIN, se simplificó significativamente la consulta. Como resultado, el costo computacional se redujo a 243 unidades y el tiempo de ejecución se acortó a 0,5 segundos. Este cambio no solo mejoró el rendimiento de la consulta sino que también simplificó el plan de ejecución, lo que facilita su mantenimiento y comprensión futura.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			69056	1576
SORT		ORDER BY	69056	1576
FILTER				
Filter Predicates				
OR				
COALESCE(E.TOTALDIAS,0)>=14				
COALESCE(C.TOTALCONSUMO,0)>15000000				
HASH JOIN		RIGHT OUTER	69056	385
Access Predicates				
U.ID=C.IDUSUARIO(+)				
VIEW			1	145
HASH		GROUP BY	1	145
HASH JOIN			3424	144
Access Predicates				
C.ID=RESERVAS.IDCONSUMO				
TABLE ACCESS	CONSUMOS	FULL	3424	49
Filter Predicates				
C.FECHA>=ADD_MONTHS(SYSDATE@!,(-12))				
TABLE ACCESS	RESERVAS	FULL	69000	95
HASH JOIN		RIGHT OUTER	69056	240
Access Predicates				
U.ID=E.IDUSUARIO(+)				
VIEW			1	97
HASH		GROUP BY	1	97
TABLE ACCESS	RESERVAS	FULL	3424	96
Filter Predicates				
AND				
R.FECHAINICIO>=ADD_MONTHS(SYSDATE@!,(-12))				
FECHAFIN>ADD_MONTHS(SYSDATE@!,(-12))				
TABLE ACCESS	USUARIOS	FULL	69056	143

Figura 7.1 Plan de ejecución de la consulta 7 generada por Oracle.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	243
SORT		ORDER BY	1	243
HASH JOIN			1	242
Access Predicates				
U.ID=C.IDUSUARIO				
Filter Predicates				
OR				
COALESCE(E.TOTALDIAS,0)>=14				
COALESCE(C.TOTALCONSUMO,0)>15000000				
NESTED LOOPS			1	97
NESTED LOOPS			1	97
VIEW			1	97
HASH		GROUP BY	1	97
TABLE ACCESS	RESERVAS	FULL	3424	96
Filter Predicates				
AND				
R.FECHAINICIO>=ADD_MONTHS(SYSDATE@!,(-12))				
FECHAFIN>ADD_MONTHS(SYSDATE@!,(-12))				
INDEX	USUARIOS_PK	UNIQUE SCAN	1	0
Access Predicates				
U.ID=E.IDUSUARIO				
TABLE ACCESS	USUARIOS	BY INDEX ROWID	1	0
VIEW			1	145
HASH		GROUP BY	1	145
HASH JOIN			3424	144
Access Predicates				
C.ID=RESERVAS.IDCONSUMO				
TABLE ACCESS	CONSUMOS	FULL	3424	49
Filter Predicates				
C.FECHA>=ADD_MONTHS(SYSDATE@!,(-12))				
TABLE ACCESS	RESERVAS	FULL	69000	95

Figura 7.2 Plan de ejecución de la consulta 7 optimizada generada por Oracle.

8. Requerimiento Funcional 8

Esta consulta original se demora 0,3 segundos en ejecutarse, por su cuenta esta realiza JOINS entre las tablas servicios, contiene y consumos utilizando el atributo id de los servicios, junto con el idservicio de la tabla contiene y idconsumo de esta misma tabla con el id de consumo. Oracle creó automáticamente dos índices: un índice en la tabla servicios para mejorar la eficiencia del bucle anidado con la tabla contiene, así como uno en la tabla contiene con la tabla consumo.

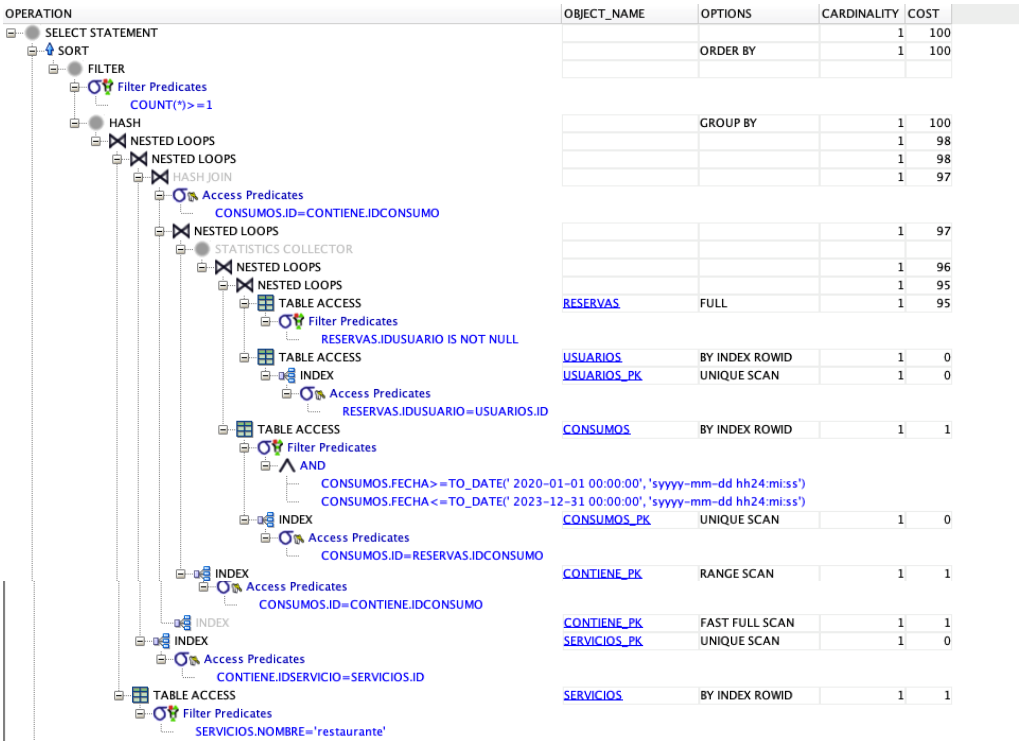
OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			49	84
FILTER				
Filter Predicates				
COUNT(*)<3				
HASH		GROUP BY	49	84
NESTED LOOPS			978	83
HASH JOIN			5708	82
Access Predicates				
CON.IDCONSUMO=C.ID				
NESTED LOOPS			5708	82
STATISTICS COLLECTOR				
TABLE ACCESS	CONSUMOS	FULL	5708	49
Filter Predicates				
C.FECHA>=ADD_MONTHS(SYSDATE@!,(-12))				
INDEX	CONTIENE_PK	RANGE SCAN	1	33
Access Predicates				
CON.IDCONSUMO=C.ID				
INDEX	CONTIENE_PK	FAST FULL SCAN	69000	33
INDEX	SERVICIOS_PK	UNIQUE SCAN	1	0
Access Predicates				
S.ID=CON.IDSERVICIO				

9. Requerimiento Funcional 9

Esta consulta original se demora 0,27 segundos en ejecutarse y tiene un costo de 100. Por su cuenta esta realiza JOINS entre las tablas usuarios, reservas, contiene, consumos y servicios entre sus id’s correspondientes relacionados. Oracle creó automáticamente cinco índices, los cuales optimizan la principalmente la relación entre las tablas en este caso:

- Index Hash Join específicamente en la tabla "contiene" a través de su índice primario para realizar la unión de manera eficiente. Siendo útil para buscar una coincidencia entre esta tabla y otra en la consulta, acelerando la búsqueda y recuperación de datos.
- Index Nested Loop: Bucle anidado con indices.
  a. Hay un índice "servicios_pk", el cual se utiliza para acelerar la operación de unión entre las tablas "contiene" y "servicios" en la consulta. El índice se basa en el predicado de acceso "contiene.idservicio = servicios.id", lo que significa que se utiliza para buscar registros en la tabla "servicios" cuyos valores de clave primaria coincidan con los valores de la columna "idservicio" en la tabla "contiene".
  b. Otro es el índice "contiene_pk" se utilizó para acelerar una operación de búsqueda y unión entre las tablas "consumos" y "contiene". El predicado de acceso "consumos.id = contiene.idconsumo" indica que el índice se utiliza para buscar registros en la tabla "consumos" cuyos valores de clave primaria coincidan con los valores de la columna "idconsumo" en la tabla "contiene". Además, se utiliza una búsqueda basada en un rango ("range scan") en el índice.
  c. El índice "consumos_pk" se implementó para acelerar las operaciones de búsqueda y unión entre las tablas "consumos" y "reservas". El predicado de acceso "consumos.id = reservas.idconsumo" indica que el índice se utiliza para buscar registros en la tabla "consumos" cuyos valores de clave primaria coincidan con los valores de la columna "idconsumo" en la tabla "reservas". Además, se utilizó una búsqueda única en el índice.

d. Se generó automáticamente también el índice "usuarios_pk" utilizado para acelerar una operación de búsqueda y unión entre las tablas "reservas" y "usuarios". El predicado de acceso "reservas.idusuario = usuarios.id" indica que el índice se utiliza para buscar registros en la tabla "usuarios" cuyos valores de clave primaria coincidan con los valores de la columna "idusuario" en la tabla "reservas". Además, se utiliza una búsqueda única en el índice.



10. Requerimiento Funcional 10

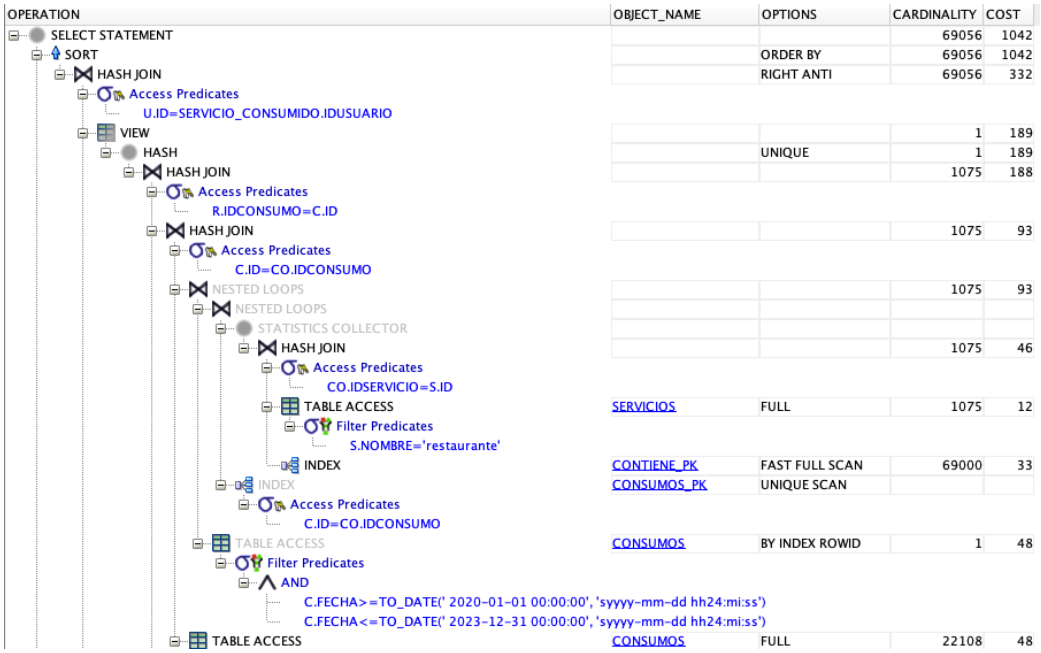
Esta consulta original se demora 1,8 segundos en ejecutarse y tiene un costo de 1042. En cuando a su desglose, la consulta principal se inicia desde la tabla "usuarios" y busca información de los usuarios. Posterior a ello, se realiza un "LEFT JOIN" con una subconsulta que busca usuarios que han consumido el servicio de restaurante en un período determinado. La subconsulta utiliza una serie de uniones para filtrar las reservas, consumos y servicios relacionados con el restaurante. La condición del "LEFT JOIN" compara los usuarios en la consulta principal con los usuarios encontrados en la subconsulta. El objetivo es encontrar usuarios que no hayan consumido el servicio de restaurante. El resultado de esta operación incluirá información de usuarios que no han consumido el restaurante, con valores NULL en las columnas relacionadas con el servicio consumido. Finalizando con una cláusula "ORDER BY" para ordenar el resultado por número de cédula, nombre y apellido de los usuarios.

Nota: El uso de un "LEFT JOIN" es apropiado en este caso, ya que permite que los usuarios que no han consumido el servicio de restaurante también aparezcan en el

resultado con valores NULL en las columnas relacionadas con el servicio. Esto facilita la identificación de usuarios que no han realizado ese consumo.

Oracle creó automáticamente dos índices, los cuales optimizan la principalmente la relación entre las tablas en este caso:

- Hay un index nested loop. El índice aplicado en "consumos_pk" se utiliza en con la siguiente condición: "c.id = con.idconsumo". Esto significa que el motor de la base de datos está utilizando el índice "consumos_pk" para buscar registros en la tabla "consumos" donde el valor de la columna "id" coincide con el valor de la columna "idconsumo" en la tabla "contiene". Junto con la opción de "unique scan" la cual indica que se está realizando una búsqueda única, dado que el índice "consumos_pk" contiene valores únicos y se está buscando una sola coincidencia.
- En la operación de hash join que involucra la tabla "contiene" y se utiliza el índice "contiene_pk", el cual está realizando un escaneo completo rápido del índice. Esto implica que se está explorando el índice de manera eficiente y rápida para realizar la operación de hash join. El índice "contiene_pk" es usado, ya que contiene información importante para la operación de unión, y el escaneo completo rápido se utiliza para garantizar un acceso eficiente a los datos requeridos durante el proceso de hash join.



11. Requerimiento Funcional 11

Esta consulta original se demora 0,206 segundos en ejecutarse y tiene un costo de 582. La consulta se divide en varias partes:
A continuación, se explica el plan de ejecución:

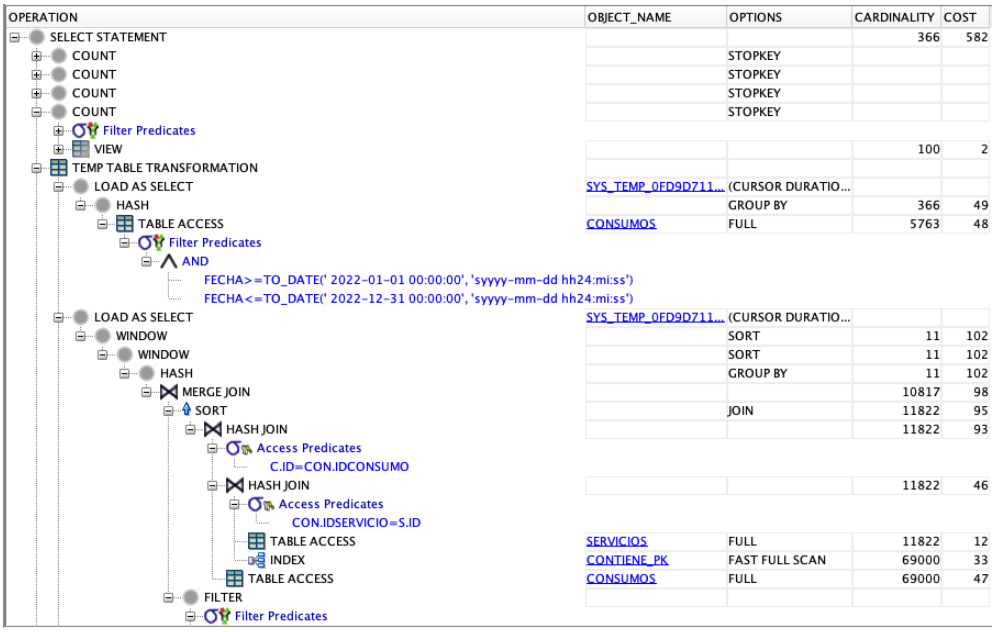
a. Semana: En esta parte, se crea una expresión común de tabla llamada "Semana" que genera un conjunto de fechas de inicio y fin de semana para cada semana del

año 2022. Esto se logra seleccionando las fechas de consumo de la tabla "consumos" y agrupándolas por semanas.

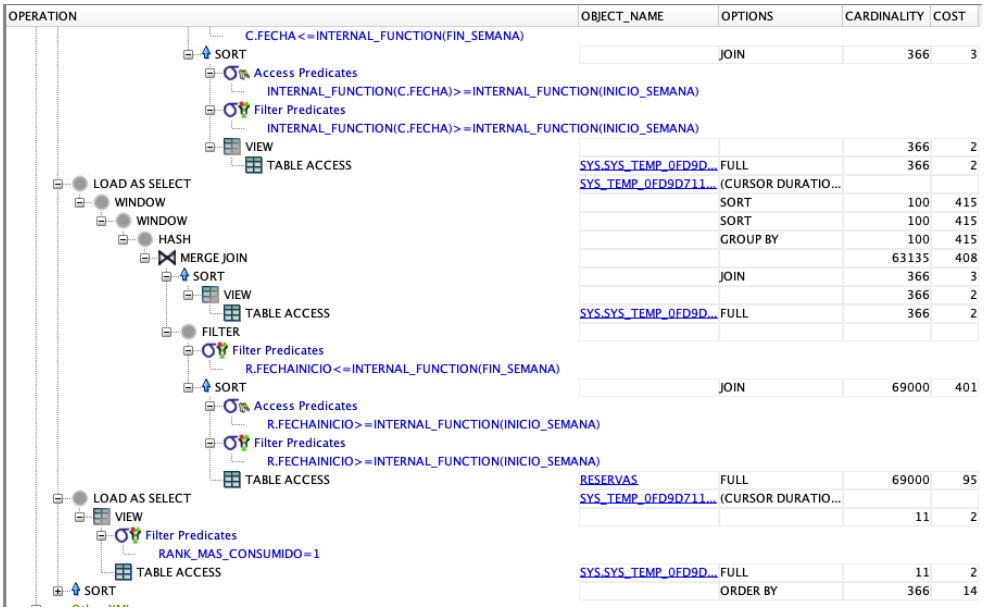
b. ServicioConsumo: Esta llamada "ServicioConsumo", calcula la cantidad de consumos de cada tipo de servicio por semana. También calcula el rango de servicios más y menos consumidos utilizando la función de ventana RANK(). Se unen las tablas "Semana," "consumos," "contiene," y "servicios" para obtener esta información.
c. HabitacionSolicitud: Esta expresión calcula la cantidad de veces que cada habitación se ha solicitado por semana. Al igual que en el "ServicioConsumo", se calculan los rangos de habitaciones más y menos solicitadas usando RANK(). Se unen las tablas "Semana" y "reservas" para obtener esta información.
d. MasConsumido y MenosConsumido: Estas ultima expresiones, identifican el servicio más consumido y menos consumido para cada semana. Esto se hace seleccionando los servicios con un rango de 1 en las expresiones anteriores y excluyendo las semanas en las que el servicio más consumido también es el servicio menos consumido.

En cuanto a la consulta principal, se unen los resultados de todas las expresiones anteriores para generar un informe que muestra la semana, el rango de fechas, el servicio más consumido y menos consumido, así como la habitación más solicitada y menos solicitada. Esto se hace utilizando subconsultas (SELECT) para obtener los valores correctos. El plan de ejecución implica el acceso y unión de las tablas y el uso de funciones analíticas (como RANK()) para calcular los rangos de los servicios y las habitaciones más y menos consumidos y solicitados.

En este caso no se generaron índices automáticos.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			366	582
COUNT		STOPKEY		
COUNT		STOPKEY		
COUNT		STOPKEY		
COUNT		STOPKEY		
Filter Predicates				
VIEW			100	2
TEMP TABLE TRANSFORMATION				
LOAD AS SELECT	SYS_TEMP_0FD9D711...	(CURSOR DURATIO...		
HASH		GROUP BY	366	49
TABLE ACCESS	CONSUMOS	FULL	5763	48
Filter Predicates				
AND				
FECHA>=TO_DATE(' 2022-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')				
FECHA<=TO_DATE(' 2022-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss')				
LOAD AS SELECT	SYS_TEMP_0FD9D711...	(CURSOR DURATIO...		
WINDOW		SORT	11	102
WINDOW		SORT	11	102
HASH		GROUP BY	11	102
MERGE JOIN			10817	98
SORT		JOIN	11822	95
HASH JOIN			11822	93
Access Predicates				
C.ID=CON.IDCONSUMO				
HASH JOIN			11822	46
Access Predicates				
CON.IDSERVICIO=S.ID				
TABLE ACCESS	SERVICIOS	FULL	11822	12
INDEX	CONTIENE_PK	FAST FULL SCAN	69000	33
TABLE ACCESS	CONSUMOS	FULL	69000	47
FILTER				
Filter Predicates				



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
C.FECHA<=INTERNAL_FUNCTION(FIN_SEMANA)				
SORT		JOIN	366	3
Access Predicates				
INTERNAL_FUNCTION(C.FECHA)>=INTERNAL_FUNCTION(INICIO_SEMANA)				
Filter Predicates				
INTERNAL_FUNCTION(C.FECHA)>=INTERNAL_FUNCTION(INICIO_SEMANA)				
VIEW			366	2
TABLE ACCESS	SYS.SYS_TEMP_0FD9D...	FULL	366	2
LOAD AS SELECT	SYS_TEMP_0FD9D711...	(CURSOR DURATIO...		
WINDOW		SORT	100	415
WINDOW		SORT	100	415
HASH		GROUP BY	100	415
MERGE JOIN			63135	408
SORT		JOIN	366	3
VIEW			366	2
TABLE ACCESS	SYS.SYS_TEMP_0FD9D...	FULL	366	2
FILTER				
Filter Predicates				
R.FECHAINICIO<=INTERNAL_FUNCTION(FIN_SEMANA)				
SORT		JOIN	69000	401
Access Predicates				
R.FECHAINICIO>=INTERNAL_FUNCTION(INICIO_SEMANA)				
Filter Predicates				
R.FECHAINICIO>=INTERNAL_FUNCTION(INICIO_SEMANA)				
TABLE ACCESS	RESERVAS	FULL	69000	95
LOAD AS SELECT	SYS_TEMP_0FD9D711...	(CURSOR DURATIO...		
VIEW			11	2
Filter Predicates				
RANK_MAS_CONSUMIDO=1				
TABLE ACCESS	SYS.SYS_TEMP_0FD9D...	FULL	11	2
SORT		ORDER BY	366	14

Carga de datos

Para la carga de datos masivos en el sistema de HotelAndes, se optó por utilizar Python, una elección estratégica debido a su eficiencia y las potentes librerías disponibles para la manipulación de datos. Específicamente, se empleó la librería Faker para generar datos ficticios que simulan información real de usuarios, como nombres y apellidos, y la librería Random para crear valores numéricos aleatorios necesarios para campos como precios o cantidades.

Se estableció un volumen de 69,000 registros para cada tabla principal, incluyendo usuarios, tiene, reservas y consumos, basándonos en la capacidad de almacenamiento en memoria de uno de los computadores del equipo, que marcaba el límite superior de lo que podíamos manejar sin comprometer el rendimiento. Esta cantidad asegura una muestra representativa para pruebas de rendimiento, manteniendo la integridad y coherencia de los datos a través de identificadores únicos y crecientes.

En el caso de la tabla de hoteles, se generó un solo registro, reflejando la singularidad del hotel en el sistema, mientras que para la tabla de habitaciones se limitó a 100 registros, asumiendo un tamaño moderado para el HotelAndes. Este enfoque permitió centrarse en la calidad de los datos más que en la cantidad, facilitando la gestión y el análisis posterior.

Uno de los desafíos más notables fue asegurar que las habitaciones no se duplicaran en las reservas para una misma fecha. Se implementaron controles de disponibilidad mediante condicionales en Python, que iteraban entre cambiar la habitación o la fecha hasta encontrar una combinación válida. Este proceso fue el más exigente en términos de tiempo de ejecución, pero crucial para mantener la lógica del negocio.

Para adaptarse a ciertos requerimientos funcionales, se realizaron ajustes manuales en los datos de las reservas, permitiendo simular escenarios donde los clientes hacían múltiples

reservas a lo largo del tiempo. Esto fue esencial para evaluar métricas clave como la fidelidad del cliente y patrones de consumo.

Pruebas de los Requerimientos

Los requerimientos se probaron en SQL Developer directamente sustituyendo los parámetros por los valores necesarios especificados. Se muestra los resultados de ejecución en tiempo para los requerimientos, recordando que el objetivo era que todas las consultas tardaran menos de 0.8 segundos.

Requerimiento	Tiempo (s)	Especificación de Prueba
RFC1	0.055	N.A.
RFC2	0.04	fechaInicio = '01-01-2016', fechafin = '31-12-2019'
RFC3	0.042	N.A.
RFC4	0.064	WHERE (c.valortotal BETWEEN 100000 AND 5000000) -- Rango de precio AND (c.fecha BETWEEN '01-01-2016' AND '31-12-2019') -- Rango de tiempo AND (s.clase = 'bar' OR s.nombre = 'bar'); -- Tipo o categoría de servicio
RFC5	0.019	(c.fecha between '01-01-2016' and '31-12-2019') and u.id = 1852
RFC6a	0.025	N.A.
RFC6b	0.02	N.A.
RFC6c	0.022	N.A.
RFC7	0.62	N.A. Nótese que el tiempo de ejecución ha aumentado
RFC8	0.236	N.A.
RFC9	0.122	N.A.
RFC10	0.092	WHERE c.fecha BETWEEN '01-01-2016' AND '31-12-2019' AND s.nombre = 'restaurante'
RFC11	0.197	N.A.