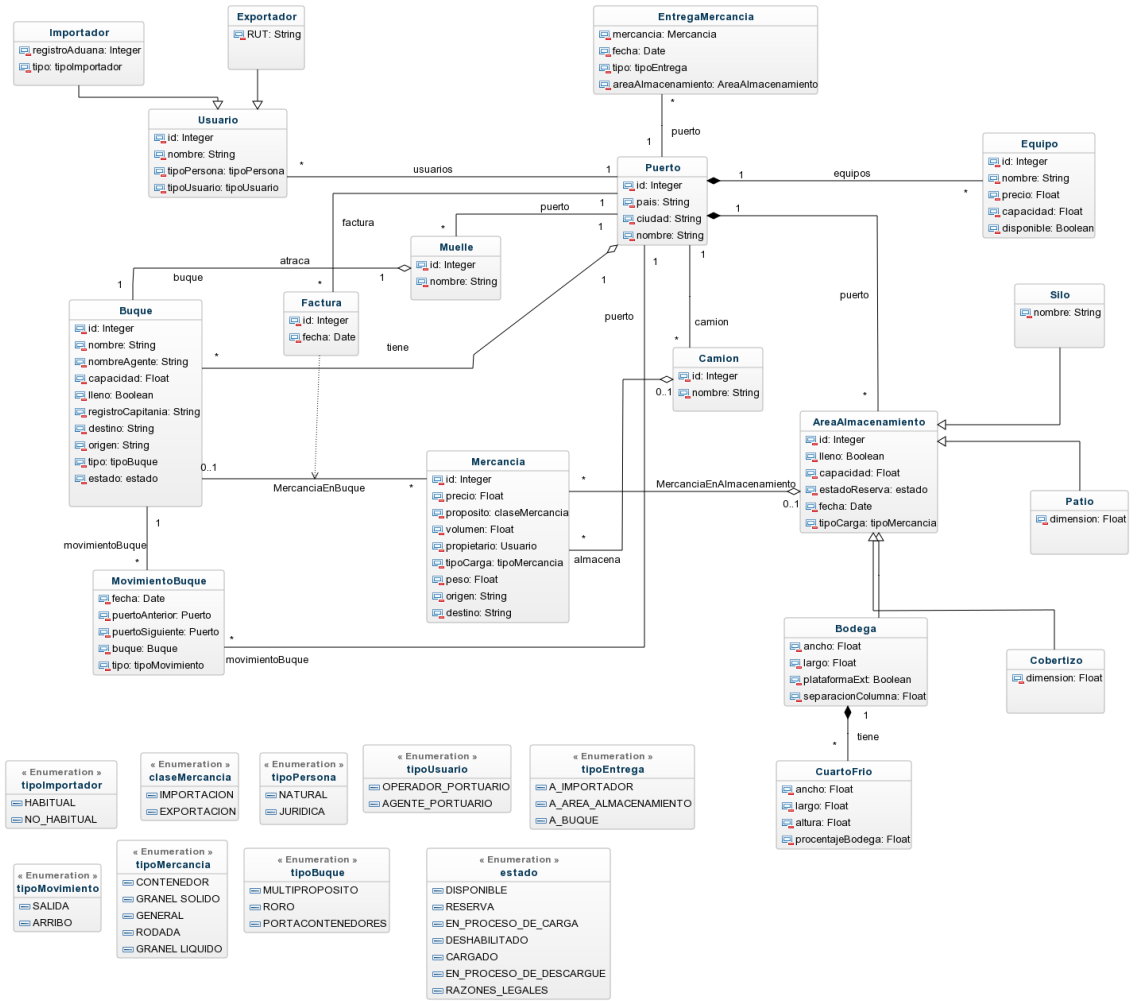
Iteración 4
Sistemas Transaccionales
Harold Gonzalez (201213646), Camilo Mendoza (201218124)
Grupo 8
01 de mayo de 2016

Análisis y Diseño de la aplicación



El Diagrama UML no requirió de modificaciones puesto que en la iteración 3 se actualizó de acuerdo a los requerimientos y para la iteración 4 las consultas se pueden realizar con el modelo actual.

Diseño físico

Índices generados por ORACLE

Los índices generados automáticamente por Oracle son de tipo B+ (default) de acuerdo a las definiciones de Oracle, no son secundarios ya que todos son únicos al ser claves primarias o definidos como únicos en las restricciones de las tablas. Los índices B+ filtran la

información de manera más sencilla, porque permiten acceder a los registros de manera más rápida.

INDEX_NAME	INDEX_TYPE	TABLE_NAME	SECONDARY
PK_ID_USUARIO	NORMAL	USUARIOS	N
PK_TIPO_IMPORTADOR	NORMAL	TIPO_IMPORTADOR	N
PK_I_ID_USUARIO	NORMAL	IMPORTADORES	N
PK_E_ID_USUARIO	NORMAL	EXPORTADORES	N
UQ_E_RUT	NORMAL	EXPORTADORES	N
PK_ID_FACTURA	NORMAL	FACTURAS	N
PK_CLASE_MERCANCIA	NORMAL	CLASE_MERCANCIA	N
PK_ID_MERCANCIA	NORMAL	MERCANCIAS	N
PK_MF_MERCANCIAS_FACTURAS	NORMAL	MERCANCIAS_ASOCIADAS_FACTURAS	N
PK_ID_MUELLE	NORMAL	MUELLES	N
PK_ID_AREA	NORMAL	AREAS_ALMACENAMIENTO	N
PK_ID_EQUIPO	NORMAL	EQUIPOS	N
PK_ID_BODEGA	NORMAL	BODEGAS	N
PK_ID_SILO	NORMAL	SILOS	N
PK_ID_COBERTIZO	NORMAL	COBERTIZOS	N
PK_ID_PATIO	NORMAL	PATIOS	N
PK_ID_CUARTO	NORMAL	CUARTOS_FRIOS	N
PK_TIPO_BUQUE	NORMAL	TIPO_BUQUE	N
PK_ID_BUQUE	NORMAL	BUQUES	N
PK_ID_CAMION	NORMAL	CAMIONES	N
PK_TIPO_USUARIO	NORMAL	TIPO_USUARIO	N
PK_TIPO_PERSONA	NORMAL	TIPO_PERSONA	N
PK_T_TIPO	NORMAL	TIPO_MERCANCIA	N
PK_TI_TIPO	NORMAL	TIPO_MOVIMIENTO	N
PK_TIPO	NORMAL	TIPO_ENTREGA	N
PK_EN	NORMAL	ENTREGA_MERCANCIA	N
PK_MEN_MERCANCIA	NORMAL	MERCANCIA_EN_BUQUE	N
UQ_MEN_ID_MERCANCIA	NORMAL	MERCANCIA_EN_BUQUE	N
PK_MEN_MER_ALM	NORMAL	MERCANCIA_EN_ALMACENAMIENTO	N
UQ_MEN_ALM_ID_MERCANCIA	NORMAL	MERCANCIA_EN_ALMACENAMIENTO	N
PK_MEN_MER_CAM	NORMAL	MERCANCIA_EN_CAMION	N
UQ_MEN_CAM_ID_MERCANCIA	NORMAL	MERCANCIA_EN_CAMION	N
PK_MOV	NORMAL	MOVIMIENTO_BUQUES	N
PK_ESTADO	NORMAL	ESTADO	N

Diseño físico por requerimientos

RFC7

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2012','DD-MM-YYYY')
```

```
            AND TO_DATE('10-02-2016','DD-MM-YYYY')
AND ID_BUQUE IN (SELECT ID_BUQUE
                 FROM BUQUES
                 WHERE NOMBRE = 'Barco66'
                   AND TIPO_BUQUE = 'RORO');
```

Análisis

Para observar el comportamiento de esta operación se utilizaron dos parámetros diferentes. Uno era un rango grande entre fechas y el otro uno muy corto, se considera grande cuando la diferencia de años es >= 2 y corto de lo contrario. También se hicieron pruebas utilizando un índice compuesto (Tipo_buque, nombre), solo un índice (fecha ó nombre ó tipo_buque) y sin índice . A continuación se muestran cada una de las conclusiones y el índice adecuado para optimizar la consulta:

- ***Sin índice y con rango de fecha grande***

Sin índice el optimizador opta por hacer un hash join con un nested loop dentro. El nested loop realiza la unión entre el acceso total de la tabla BUQUES y un escaneo de rango con la PK_MOV (de la tabla MOVIMIENTO_BUQUES), cuando accede a la tabla busca por medio

de un filtro el nombre y el tipo de buque, luego en el escaneo por rango busca las fechas y el id del buque utilizando un filtro en el id_buque. Finalmente hace un hash join con el resultado del nested loop y el acceso total de la tabla MOVIMIENTO_BUQUES con un filtro en las dos fechas. Estas operaciones tienen una cardinalidad total de 7757 y costo de 4945 y el tiempo de ejecución del plan fue de 0.032 como se puede apreciar en la imagen. Más adelante se compararán estos resultados con la implementación de índices para definir si existen mejoras o no. La razón por la que hace hash join se debe a que los datos no se encuentran ordenados, de lo contrario sería más eficiente un merge sort.

- ***Sin índice y con rango de fecha pequeño***

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2016','DD-MM-YYYY')
  AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE IN (SELECT ID_BUQUE
                   FROM BUQUES
                   WHERE NOMBRE = 'Barco66'
                     AND TIPO_BUQUE = 'RORO');
```

Salida de Script | Resultado de la Consulta | Explicación del Plan

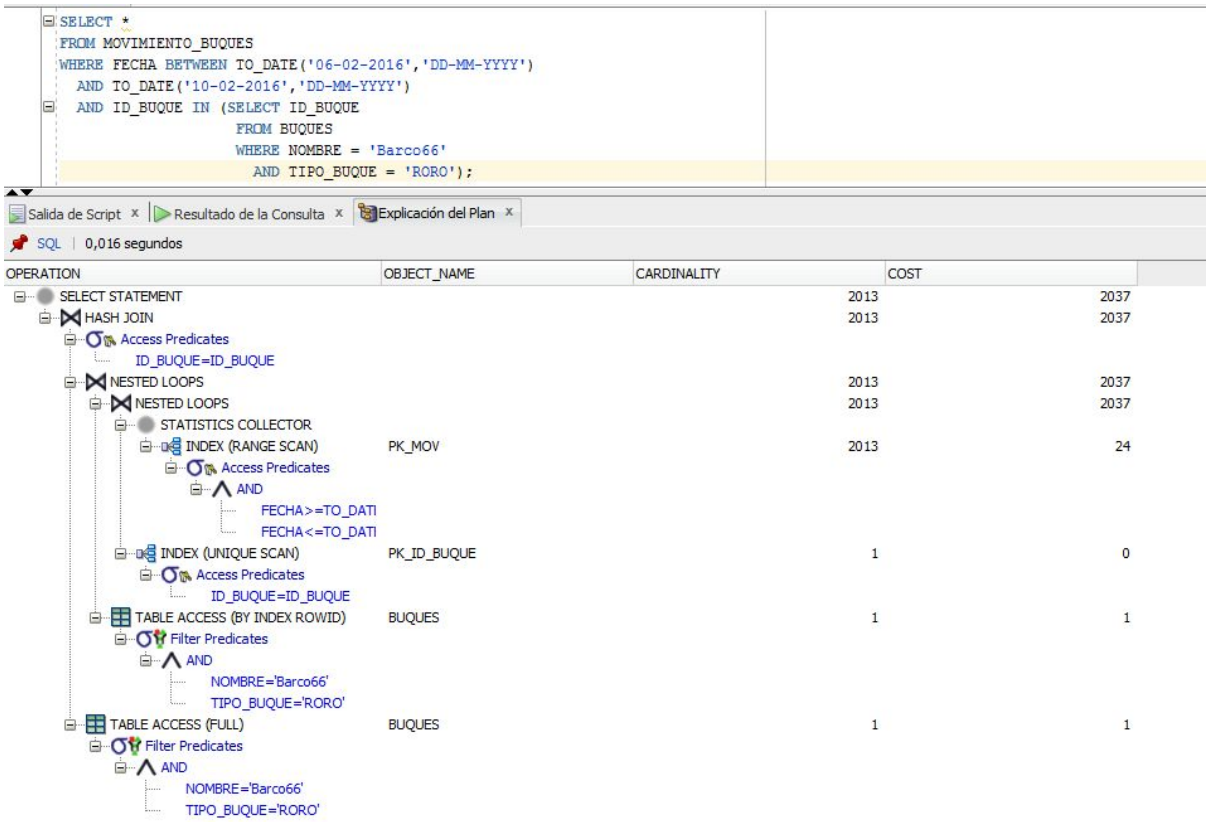
SQL | 0,001 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1895	1918
HASH JOIN		1895	1918
Access Predicates			
ID_BUQUE=ID_BUQUE			
NESTED LOOPS		1895	1918
NESTED LOOPS		1895	1918
STATISTICS COLLECTOR			
INDEX (RANGE SCAN)	PK_MOV	1895	23
Access Predicates			
AND			
FECHA>=TO_DATI			
FECHA<=TO_DATI			
INDEX (UNIQUE SCAN)	PK_ID_BUQUE	1	0
Access Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (BY INDEX ROWID)	BUQUES	1	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
TABLE ACCESS (FULL)	BUQUES	1	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			

Cuando en la consulta se incluye un rango de fecha corto y no se utilizan índices, el optimizador realiza un hash join con dos Nested loops en su interior, esto para ayudar al hash join con el acceso de las filas de la tabla interna. El nested loop interior (N1) realiza un join con un Range Scan del índice PK_MOV para buscar el rango de fechas y un Unique Scan del índice PK_ID_BUQUE con el ID_BUQUE indicado. Luego el Nested loop exterior (N2) realiza un join con el resultado de (N1) y un acceso por ROWID de la tabla BUQUES (debido a que hash join no puede acceder a cada una de las filas internas), aplicando un filtro sobre NOMBRE y TIPO_BUQUE. Finalmente realiza el hash join con el resultado de (N2) y el acceso total de la tabla BUQUES aplicando un filtro sobre NOMBRE y

TIPO_BUQUE. Esta operación tiene una cardinalidad total de 1895 y un costo de 1918. En este caso el optimizador utilizó dos Nested Loop porque este tipo de join es más eficiente cuando hay un rango pequeño de datos en la tabla, sin embargo se ve obligado a usar un hash join debido a que los datos no se encuentran ordenados.

- ***Utilizando un índice en un rango de fecha pequeño***



En este caso se utilizó un índice en la fecha para probar si se obtenía alguna mejora, sin embargo el optimizador no utilizó el índice por baja selectividad además no hubo diferencia en los resultados a excepción de la cardinalidad que aumentó en valor de 200 aproximadamente. Por lo anterior se considera que de esta forma la operación se vuelve un poco ineficiente que si no se usara el índice fecha. Por otro lado, también se usaron índices sobre Tipo_buque y nombre, pero los resultados fueron iguales a no usar índice.

- ***Utilizando un índice con rango de fecha grande***

- *Con índice en fecha*

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2012','DD-MM-YYYY')
    AND TO_DATE('10-02-2016','DD-MM-YYYY')
```

Salida de Script | Explicación del Plan

SQL | 0,972 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2393	4945
HASH JOIN		2393	4945
Access Predicates			
ID_BUQUE=ID_BUQUE			
NESTED LOOPS		2393	4945
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	BUQUES	6165	3301
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
INDEX (RANGE SCAN)	PK_MOV	1	2462
Access Predicates			
AND			
FECHA>=TO_DATE(' 2012-0			
ID_BUQUE=ID_BUQUE			
FECHA<=TO_DATE(' 2016-0			
Filter Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (FULL)	MOVIMIENTO_BUQUES	458752	1643
Filter Predicates			
AND			
FECHA>=TO_DATE(' 2012-02-06			
FECHA<=TO_DATE(' 2016-02-10			

Utilizando un índice en fecha el optimizador no lo utiliza, y la consulta mejora en comparación a no usar índice, pero no son los resultados más óptimos. Además el tiempo de ejecución fue de 0.972 segundos lo cual es bastante alto.

- *Con índice en Tipo_Buque*

```
    WHERE NOMBRE = 'Barco66'
      AND TIPO_BUQUE = 'RORO');
```

A141610.AREAS_ALMACENAMIENTO@Camilo

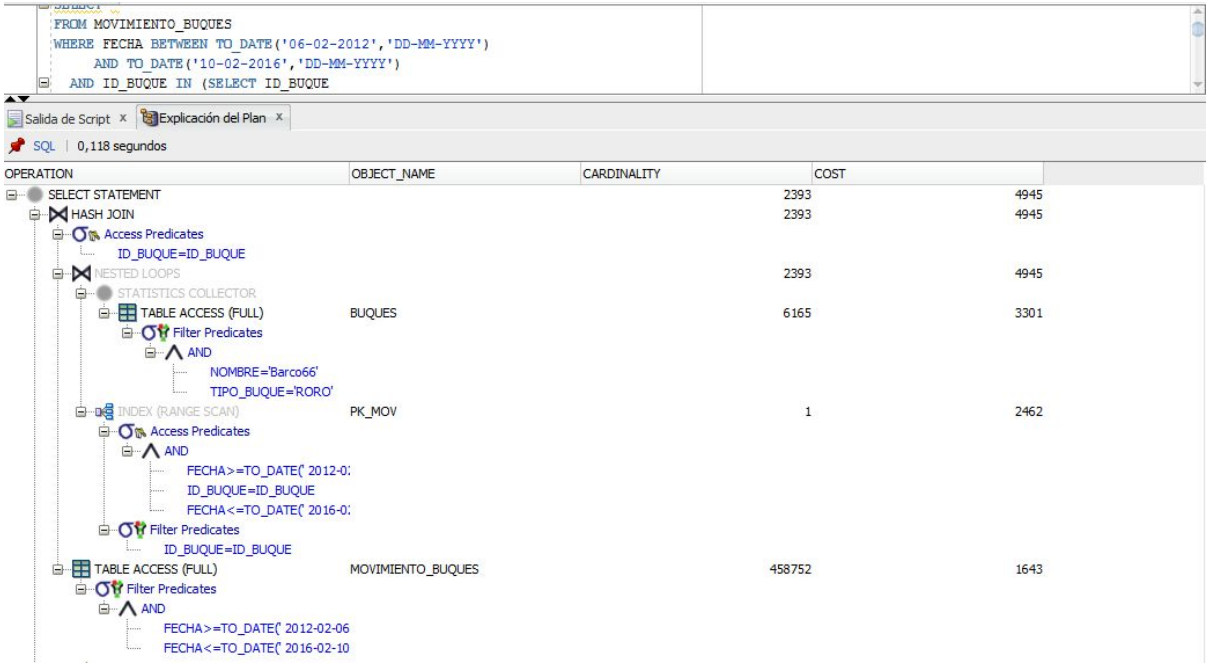
Salida de Script | Explicación del Plan

SQL | 0,121 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2393	4945
HASH JOIN		2393	4945
Access Predicates			
ID_BUQUE=ID_BUQUE			
NESTED LOOPS		2393	4945
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	BUQUES	6165	3301
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
INDEX (RANGE SCAN)	PK_MOV	1	2462
Access Predicates			
AND			
FECHA>=TO_DATE(' 2012-0			
ID_BUQUE=ID_BUQUE			
FECHA<=TO_DATE(' 2016-0			
Filter Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (FULL)	MOVIMIENTO_BUQUES	458752	1643
Filter Predicates			
AND			
FECHA>=TO_DATE(' 2012-02-06			
FECHA<=TO_DATE(' 2016-02-10			

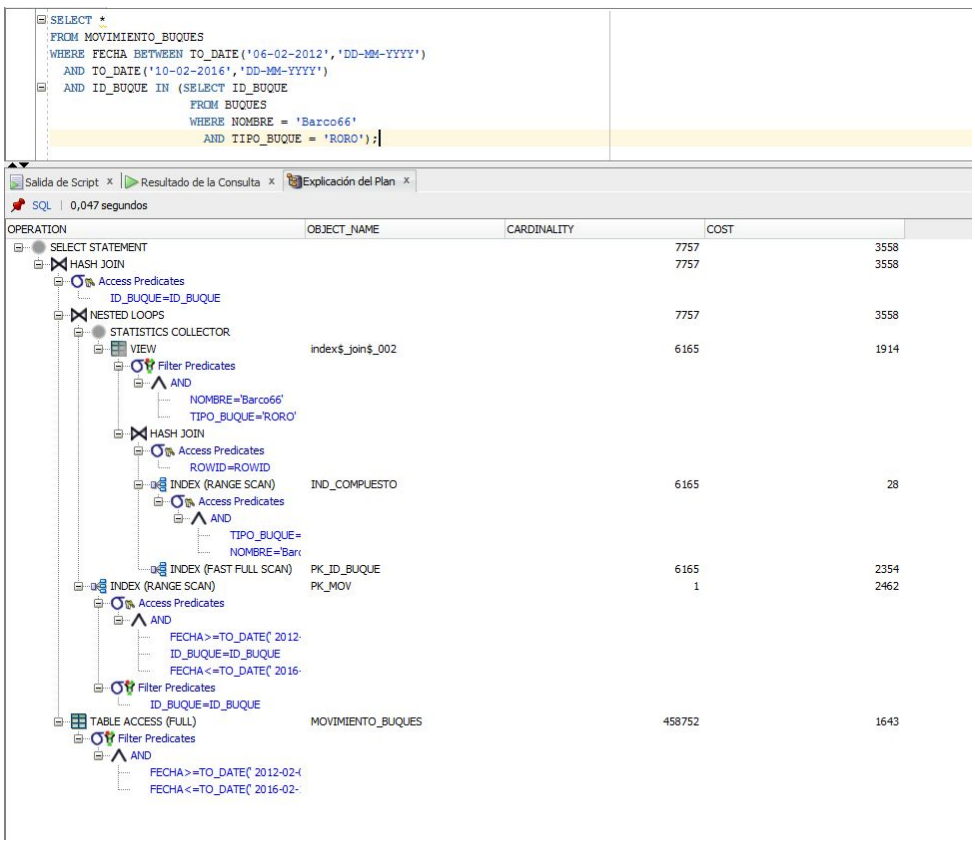
Ocurre igual que al usar índice en fecha y además el tiempo de ejecución aumenta, por lo que se descarta esta opción.

- *Con índice en NOMBRE*



Igual que en los casos anteriores y el tiempo de ejecución aumenta. Se descarta también esta opción.

- ***Utilizando un índice compuesto (Tipo_buque, Nombre) con un rango de fecha grande***



En comparación al uso de un índice y no índice, al utilizar un índice compuesto en la tabla BUQUES, el optimizador hace un hash join (HJ2) con un nested loop (N1) que contiene un hash join a su vez (HJ1).
HJ2 se realiza para poder ubicar los índices mediante ROWID con un Range Scan aplicando el filtro sobre el TIPO_BUQUE Y NOMBRE, elementos que conforman el índice compuesto y el join con fast full scan de la PK_ID_BUQUE, de esta manera el hash join puede leer las filas de la tabla interior sin ayuda de un nested loop. Ahora se realiza el N1 con el resultado de HJ2 join Range Scan de las fechas y el id_buque aplicando filtro sobre ID_BUQUE, de esta manera se confirma que exista un buque con el id. Finalmente se realiza HJ2 con acceso total a la tabla MOVIMIENTO_BUQUES con filtro del rango de fechas. La cardinalidad total es de 7757 y el costo de 3558.

En comparación con los casos en donde se utiliza un índice y donde no, este caso de índice compuesto resulta ser el mejor para un rango de fecha grande debido a que los costos son menores, disminuyen de 4945 a 3558 aún con la misma cardinalidad y la diferencia del tiempo de ejecución es mínima, tan solo aumenta 0.010 segundos comparado con el menor tiempo.

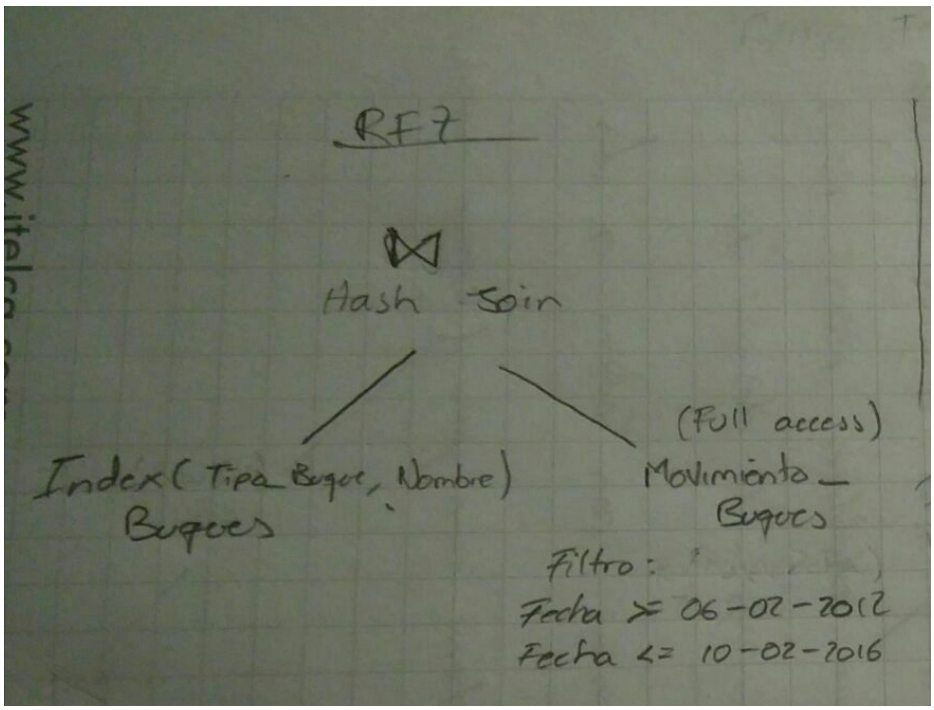
Este índice no presenta mayor problema de mantenimiento pues los tipos de buque no cambian en mucho tiempo y, aunque pueden agregarse nuevos buques a la base de datos, los buques también se mantienen relativamente estables, aunque puede ocupar relativamente bastante espacio en disco.

- ***Utilizando índice compuesto (Tipo_buque, Nombre) con rango pequeño***

En este caso el optimizador solo realiza un doble nested loop debido a que la cantidad de datos consultados tiene un tamaño muy pequeño. Ahora comparando este caso con los

otros donde se usaba un índice y donde no, se evidencia que la cardinalidad total disminuye considerablemente pues es tan solo de 8 y el costo se mantiene igual a no usar un índice. Así entonces se considera que **la opción más eficiente es usar un índice compuesto**.

Plan propuesto antes de ejecución sql (RF7)



Comparación planes de ejecución

Mientras nosotros usamos un hash join entre las dos tablas, Oracle usa también un hash join pero con un nested loop dentro para ayudar al hash en las lecturas de las filas de la tabla interna.

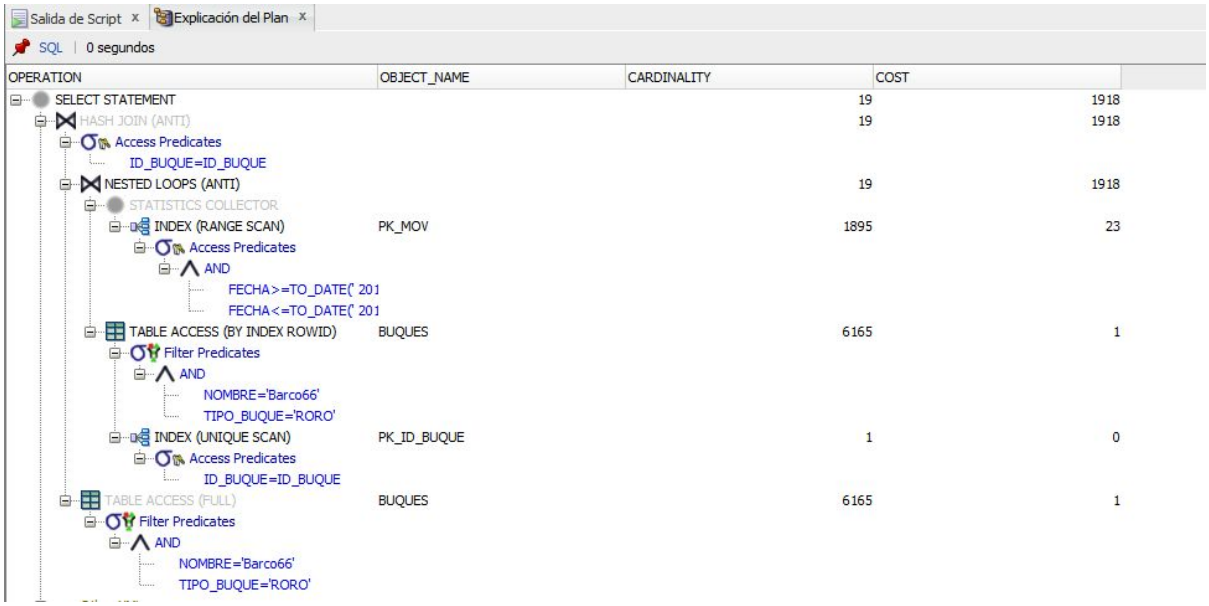
RFC8

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2016','DD-MM-YYYY')
              AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE NOT IN (SELECT ID_BUQUE
                       FROM BUQUES
                       WHERE NOMBRE = 'Barco66'
                         AND TIPO_BUQUE = 'RORO');
```

Análisis:

Para observar el comportamiento de esta operación se utilizaron dos parámetros diferentes. Uno era un rango grande entre fechas y el otro uno muy corto, se considera grande cuando la diferencia de años es >= 2 y corto de lo contrario. También se hicieron pruebas utilizando un índice compuesto (Tipo_buque, nombre), solo un índice (fecha o nombre o tipo_buque) y sin índice . A continuación se muestran cada una de las conclusiones y el índice adecuado para optimizar la consulta:

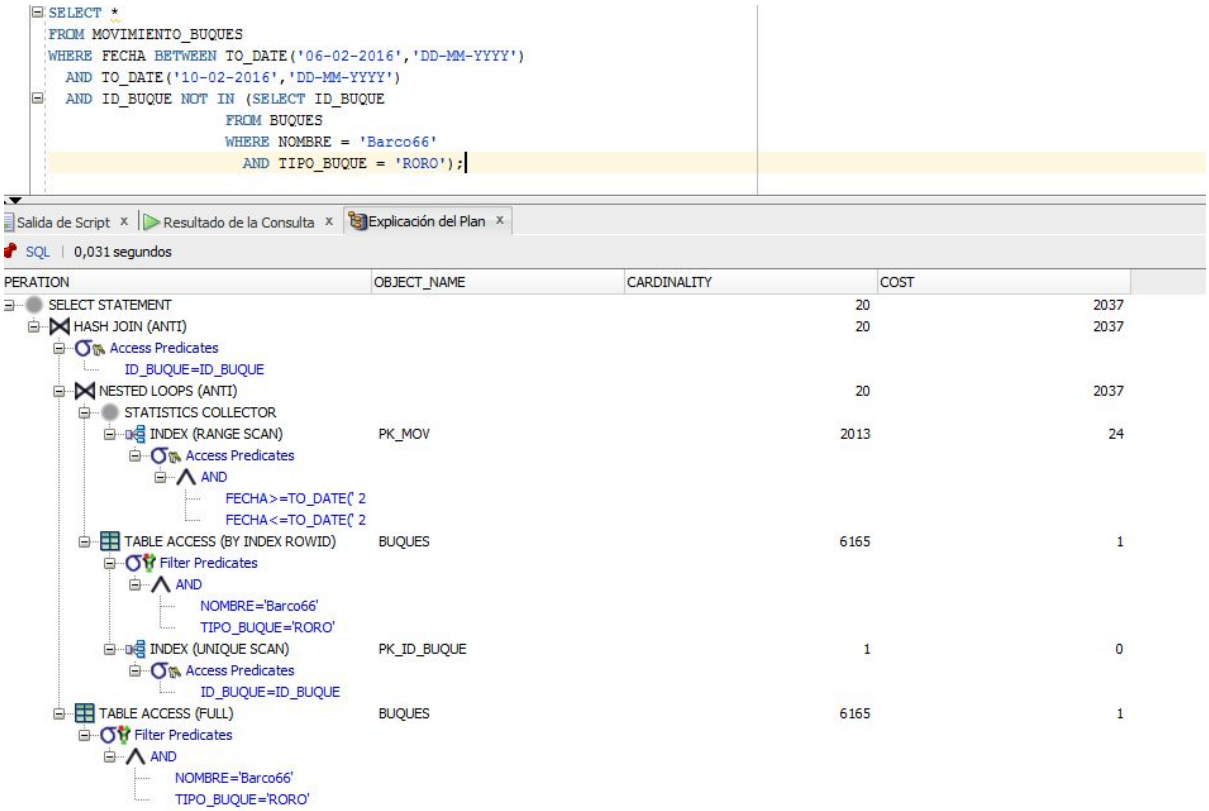
- ***Sin índice y con rango de fecha corto***



En este caso el optimizador hace un anti hash join ya que se utiliza NOT IN en la sentencia sql, a su vez este contiene un anti nested loop para ayudar al hash con las lecturas de las filas de la tabla interna. El loop hace un join entre el range scan utilizando como predicados el rango de fecha y el acceso a la tabla BUQUES por ROWID aplicando el filtro en nombre y tipo_buque y un unique scan al mismo. Luego se hace el hash join con el resultado del loop y el acceso total de la tabla BUQUES aplicando el filtro sobre nombre y tipo_buque teniendo como predicado el id del buque. La cardinalidad de esta operación es 19 y el costo es de 1918.

- ***Utilizando un índice en un rango de fecha corto***

- *Con índice en fecha*



En comparación con el caso anterior este es menos eficiente ya que el optimizador realiza el mismo proceso, pero la cardinalidad aumenta al igual que el costo. Por lo tanto no es una buena estrategia usar un índice en fecha.

- *Con índice en Tipo_buque*

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2016','DD-MM-YYYY')
  AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE NOT IN (SELECT ID_BUQUE
                       FROM BUQUES
                       WHERE NOMBRE = 'Barco66'
                         AND TIPO_BUQUE = 'RORO');

CREATE INDEX IND_TIPO_BUQUE ON BUQUES(TIPO_BUQUE);
```

Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0,063 segundos

PERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		19	1918
HASH JOIN (ANTI)		19	1918
Access Predicates			
ID_BUQUE=ID_BUQUE			
NESTED LOOPS (ANTI)		19	1918
STATISTICS COLLECTOR			
INDEX (RANGE SCAN)	PK_MOV	1895	23
Access Predicates			
AND			
FECHA>=TO_DATE(' 2			
FECHA<=TO_DATE(' 2			
TABLE ACCESS (BY INDEX ROWID)	BUQUES	6165	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
INDEX (UNIQUE SCAN)	PK_ID_BUQUE	1	0
Access Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (FULL)	BUQUES	6165	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			

En este caso el optimizador no utiliza el índice debido a baja selectividad y por lo tanto la operación realiza el mismo procedimiento a no usar un índice.

- *Con índice en Nombre*

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2016','DD-MM-YYYY')
  AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE NOT IN (SELECT ID_BUQUE
                       FROM BUQUES
                       WHERE NOMBRE = 'Barco66'
                         AND TIPO_BUQUE = 'RORO');
```

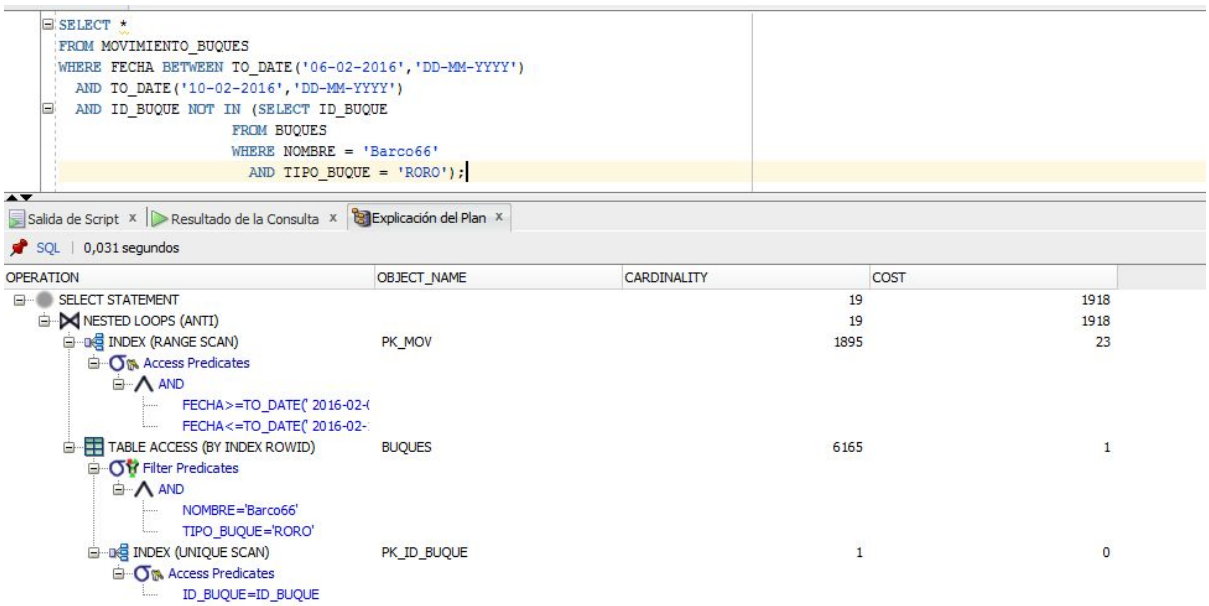
Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		19	1918
HASH JOIN (ANTI)		19	1918
Access Predicates			
ID_BUQUE=ID_BUQUE			
NESTED LOOPS (ANTI)		19	1918
STATISTICS COLLECTOR			
INDEX (RANGE SCAN)	PK_MOV	1895	23
Access Predicates			
AND			
FECHA>=TO_DATE(' 2			
FECHA<=TO_DATE(' 2			
TABLE ACCESS (BY INDEX ROWID)	BUQUES	6165	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
INDEX (UNIQUE SCAN)	PK_ID_BUQUE	1	0
Access Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (FULL)	BUQUES	6165	1
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			

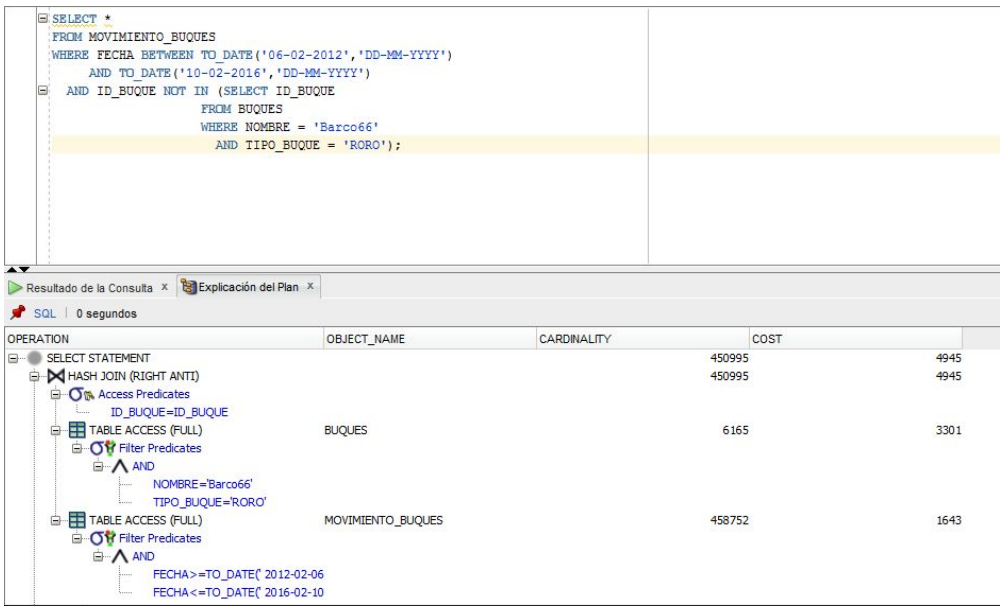
En este caso ocurre igual que al usar el índice en Tipo_buque, por lo que se concluye que no es útil usar un índice en un rango de fecha corto.

- ***Usando un índice compuesto (Tipo_Buque, Nombre) en un rango de fecha corto***



En este caso aunque el optimizador solo realiza un anti nested loop, la cardinalidad y el costo son iguales a no usar un índice, con excepción en el tiempo de ejecución pues sin índice es de 0 segundos. En conclusión al ver que usar algún tipo de índice es inútil, se decide no usar índice para consultas en un rango de fecha corto.

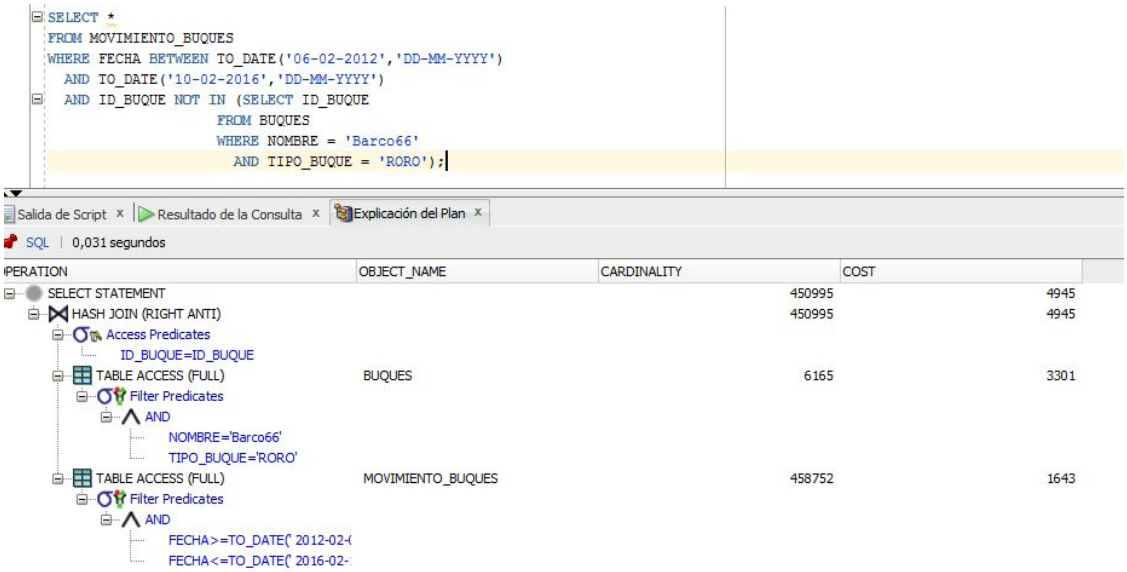
- ***Sin índice en un rango de fecha grande***



Sin índice el optimizador hace un Right anti hash join porque es el plan de ejecución con el costo más bajo cuando la tabla especificada en la sentencia NOT IN es más grande que la especificada en la sentencia FROM. El hash hace un acceso total para la tabla BUQUES aplicando un filtro en el nombre y el id_buque y un acceso total en la tabla MOVIMIENTO_BUQUES aplicando el filtro sobre el rango de fechas, teniendo como predicado el id_buque. Esta operación tiene una cardinalidad de 450995, un costo de 4945 y tiempo de ejecución en 0 segundos.

- ***Usando un índice en un rango de fecha grande***

- *Con índice en fecha*



En este caso no hay cambios en comparación a no usar índice, a excepción del tiempo de ejecución que aumenta.

- *Con índice en Tipo_buque*

Tampoco se presentan cambios en comparación a los casos anteriores, pero el tiempo de ejecución es menor a usar el índice fecha mas no a cuando no hay índice.

- *Con índice en nombre*

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2012','DD-MM-YYYY')
  AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE NOT IN (SELECT ID_BUQUE
                       FROM BUQUES
                       WHERE NOMBRE = 'Barco66'
                         AND TIPO_BUQUE = 'RORO');
```

Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0,046 segundos

PERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		450995	4945
HASH JOIN (RIGHT ANTI)		450995	4945
Access Predicates			
ID_BUQUE=ID_BUQUE			
TABLE ACCESS (FULL)	BUQUES	6165	3301
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
TABLE ACCESS (FULL)	MOVIMIENTO_BUQUES	458752	1643
Filter Predicates			
AND			
FECHA>=TO_DATE(' 2012-02-(			
FECHA<=TO_DATE(' 2016-02-			

Tampoco se presentan cambios, a excepción del tiempo de ejecución. Finalmente se concluye que un índice resulta inútil para optimizar la operación, puesto que se obtienen los mismos resultados al no usar.

- ***Utilizando un índice compuesto (Tipo_Buque, Nombre) en un rango de fecha grande***

```
SELECT *
FROM MOVIMIENTO_BUQUES
WHERE FECHA BETWEEN TO_DATE('06-02-2012','DD-MM-YYYY')
  AND TO_DATE('10-02-2016','DD-MM-YYYY')
  AND ID_BUQUE NOT IN (SELECT ID_BUQUE
                       FROM BUQUES
                       WHERE NOMBRE = 'Barco66'
                         AND TIPO_BUQUE = 'RORO');
```

Salida de Script | Resultado de la Consulta | Explicación del Plan

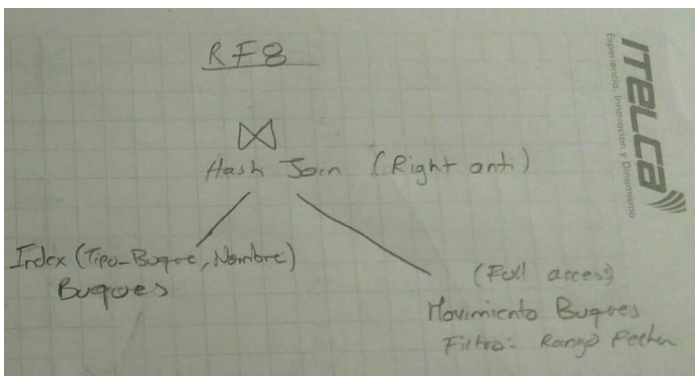
SQL | 0,047 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		450995	3558
HASH JOIN (RIGHT ANTI)		450995	3558
Access Predicates			
ID_BUQUE=ID_BUQUE			
VIEW	index$_join$_002	6165	1914
Filter Predicates			
AND			
NOMBRE='Barco66'			
TIPO_BUQUE='RORO'			
HASH JOIN			
Access Predicates			
ROWID=ROWID			
INDEX (RANGE SCAN)	IND_COMPUESTO	6165	28
Access Predicates			
AND			
TIPO_BUQUE='RORO'			
NOMBRE='Barco66'			
INDEX (FAST FULL SCAN)	PK_ID_BUQUE	6165	2354
TABLE ACCESS (FULL)	MOVIMIENTO_BUQUES	458752	1643
Filter Predicates			
AND			
FECHA>=TO_DATE(' 2012-02-(			
FECHA<=TO_DATE(' 2016-02-			

En este caso el optimizador realiza un Hash join anti con un hash join normal dentro. El hash Join de adentro realiza un range scan del índice compuesto y hace la unión con pk_id_buque. Luego con este resultado se hace un join con el acceso total de la tabla MOVIMIENTO_BUQUES. La cardinalidad de esta operación es igual que en los otros (450995), sin embargo el costo ahora se redujo a 3558 y el tiempo de ejecución es de 0.047 segundos. Por lo tanto, al ver que los costos se reducen se decide que utilizar el índice compuesto es lo más óptimo para la operación.

Este índice, como es el mismo del requerimiento anterior, no presenta problemas de mantenimiento, aunque puede ocupar relativamente bastante espacio en disco.

Plan propuesto antes de ejecución sql (RF8)



Comparación planes de ejecución

Oracle utiliza un hash join anti con un hash join dentro. El de adentro lo usa para comparar la pk con el índice compuesto y luego usa el anti para descartar la información correspondiente en la tabla movimiento_buque. Mientras en el plan de nosotros solo usamos un hash anti entre las dos tablas.

RFC9

```
SELECT B.ID_MERCANCIA, B.ORIGEN, B.DESTINO, B.TIPO_CARGA, A.FECHA_ORDEN,
A.FECHA_REALIZACION
FROM ENTREGA_MERCANCIA A
    JOIN
    MERCANCIAS B
    ON A.ID_MERCANCIA = B.ID_MERCANCIA
WHERE B.PRECIO>100
      AND B.TIPO_CARGA = 'GENERAL'
      AND B.PROPIETARIO = 1;
```

Análisis

El costo de esta operación cambia ligeramente con el tipo de carga y el propietario, pues existen pequeñas diferencias entre la selectividad de estos atributos dependiendo de su valor. Sin embargo, con el precio no presenta diferencias de costo debido a que es una búsqueda de rango. Por esta razón, para el análisis de los índices no se tendrá en cuenta la distribución de los datos

Sin índice

```
SELECT B.ID_MERCANCIA, B.ORIGEN, B.DESTINO, B.TIPO_CARGA, A.FECHA_ORDEN, A.FECHA_REALIZACION
FROM ENTREGA_MERCANCIA A
    JOIN
    MERCANCIAS B
    ON A.ID_MERCANCIA = B.ID_MERCANCIA
WHERE B.PRECIO>100
      AND B.TIPO_CARGA = 'GENERAL'
      AND B.PROPIETARIO = 1;
```

Salida de Script | Explicación del Plan

SQL | 0,02 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		19	2795
HASH JOIN		19	2795
Access Predicates			
A.ID_MERCANCIA=B.ID_MERCANCIA			
NESTED LOOPS		19	2795
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	MERCANCIAS	50	2745
Filter Predicates			
AND			
B.TIPO_CARGA='GENE			
B.PROPIETARIO=1			
B.PRECIO>100			
INDEX (RANGE SCAN)	PK_EN	1	1
Access Predicates			
A.ID_MERCANCIA=B.ID_MERC			
TABLE ACCESS (FULL)	ENTREGA_MERCANCIA	1	1

Cuando no se tienen índices la operación principal que realiza el optimizador es un hash join. Este join se hace entre el resultado de un nested loop y un acceso completo a la tabla Entrega_Mercancia. El nested loop se hace entre un acceso completo a la tabla Mercancias, que luego se filtra por los tres parámetros del requerimiento, y un range scan usando la llave primaria para realizar el join. El optimizador realiza este hash join pues los datos no están organizados por los parámetros de la consulta y existe mucha diferencia entre la cardinalidad de ambos componentes del join. Con la consulta de prueba se obtiene una cardinalidad de 19 y un costo de 2795, al cual influye mucho el acceso completo a la tabla Mercancias. La consulta no tomó mucho tiempo: solo 0.02 segundos.

Con índice compuesto (Precio, Tipo_Carga, Propietario)

```
SELECT B.ID_MERCANCIA, B.ORIGEN, B.DESTINO, B.TIPO_CARGA, A.FECHA_ORDEN, A.FECHA_REALIZACION
FROM ENTREGA_MERCANCIA A
    JOIN
    MERCANCIAS B
    ON A.ID_MERCANCIA = B.ID_MERCANCIA
WHERE B.PRECIO>100
      AND B.TIPO_CARGA = 'GENERAL'
      AND B.PROPIETARIO = 1;
```

Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0,02 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		19	50
HASH JOIN		19	50
Access Predicates			
A.ID_MERCANCIA=B.ID_MERCANCIA			
NESTED LOOPS		19	50
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWID E	MERCANCIAS	25	25
INDEX (RANGE SCAN)	I_COMPUESTO	18	3
Access Predicates			
AND			
B.PRECIO>100			
B.TIPO_CARGA='G			
B.PROPIETARIO=1			
Filter Predicates			
AND			
B.TIPO_CARGA='G			
B.PROPIETARIO=1			
INDEX (RANGE SCAN)	PK_EN	1	1
Access Predicates			
A.ID_MERCANCIA=B.ID_MERC			
TABLE ACCESS (FULL)	ENTREGA_MERCANCIA	1	1

La solución más obvia sería realizar un índice compuesto con los tres parámetros de la consulta. Este índice mejora significativamente el costo de la consulta pues, como se puede apreciar en el plan de ejecución, el acceso a la tabla Mercancias ya no se hace completo sino utilizando el índice y accediendo por el ROWID. Esto reduce el costo de 2795 a 50, con la cardinalidad y el tiempo de ejecución manteniéndose iguales.

Con índice compuesto (Tipo_Carga, Propietario)

```
SELECT B.ID_MERCANCIA, B.ORIGEN, B.DESTINO, B.TIPO_CARGA, A.FECHA_ORDEN, A.FECHA_REALIZACION
FROM ENTREGA_MERCANCIA A
    JOIN
    MERCANCIAS B
    ON A.ID_MERCANCIA = B.ID_MERCANCIA
WHERE B.PRECIO>100
      AND B.TIPO_CARGA = 'GENERAL'
      AND B.PROPIETARIO = 1;
```

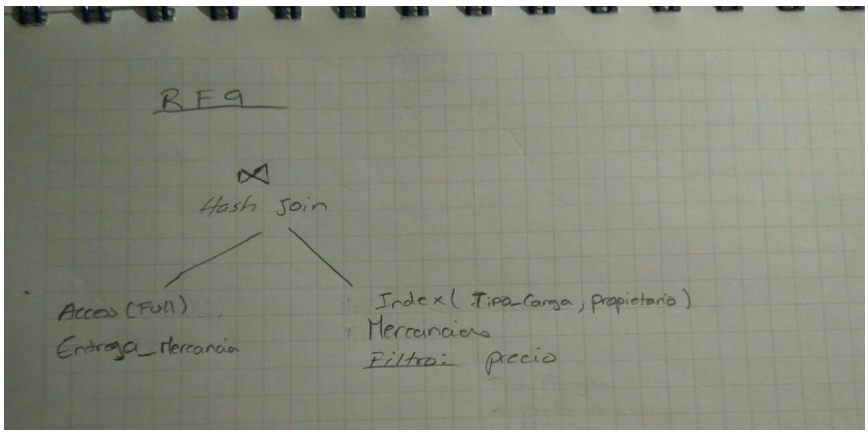
Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0,04 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		19	48
HASH JOIN		19	48
Access Predicates			
A.ID_MERCANCIA=B.ID_MERCANCIA			
NESTED LOOPS		19	48
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWID E	MERCANCIAS	25	23
Filter Predicates			
B.PRECIO>100			
INDEX (RANGE SCAN)	I_COMPUESTO	16	3
Access Predicates			
AND			
B.TIPO_CARGA='G			
B.PROPIETARIO=1			
INDEX (RANGE SCAN)	PK_EN	1	1
Access Predicates			
A.ID_MERCANCIA=B.ID_MERC			
TABLE ACCESS (FULL)	ENTREGA_MERCANCIA	1	1

Luego de realizar el análisis usando el índice compuesto con los tres parámetros, se decidió reducir este índice a solo dos parámetros: Tipo_Carga y Propietario. Estos parámetros son los que afectan, aunque muy poco, el costo y la cardinalidad pues el Precio es una búsqueda de rango y con el índice de árbol B+ no hay diferencias de costo por cambiar el tamaño de este rango. Con este índice se redujo ligeramente el costo, por lo que se decidió finalmente utilizar este índice, al reducir el tamaño en memoria que ocuparía el índice y teniendo en cuenta que estos dos parámetros no presentan muchos cambios en la vida útil de la base de datos.

Plan propuesto antes de ejecución sql (RF9)



Comparación planes de ejecución

Oracle realiza un nested loop para poder realizar el filtro de la tabla Mercancias, de acuerdo al hash join de afuera, mientras que en nuestro diseño el hash join se hace directamente entre el filtro (acceso por el índice) y un acceso completo a la tabla Entrega_Mercancia. Esto se debe a que Oracle generalmente realiza nested loops dentro de hash join para optimizar el proceso, pues así se puede acceder más fácilmente a los resultados que se buscan.

RFC10

```
SELECT DISTINCT A.*, B.ESTADO
FROM ENTREGA_MERCANCIA A
    JOIN
    AREAS_ALMACENAMIENTO B
    ON B.ID_AREA = A.ID_AREA_ALMACENAMIENTO
        OR B.ID_AREA = A.ID_AREA_ALMACENAMIENTO_2
WHERE A.ID_AREA_ALMACENAMIENTO IN (1,153201)
      OR A.ID_AREA_ALMACENAMIENTO_2 IN (1,153201);
```

Análisis

Cambiar el id de las áreas de almacenamiento que se quieren consultar no afecta el costo de esta consulta, pues las operaciones que se realizan son las mismas, con accesos a las tablas y uso de índices de la misma manera.

Sin índices

```
SELECT DISTINCT A.*, B.ESTADO
FROM ENTREGA_MERCANCIA A
    JOIN
    AREAS_ALMACENAMIENTO B
    ON B.ID_AREA = A.ID_AREA_ALMACENAMIENTO
        OR B.ID_AREA = A.ID_AREA_ALMACENAMIENTO_2
WHERE A.ID_AREA_ALMACENAMIENTO IN (1,153201)
      OR A.ID_AREA_ALMACENAMIENTO_2 IN (1,153201);
```

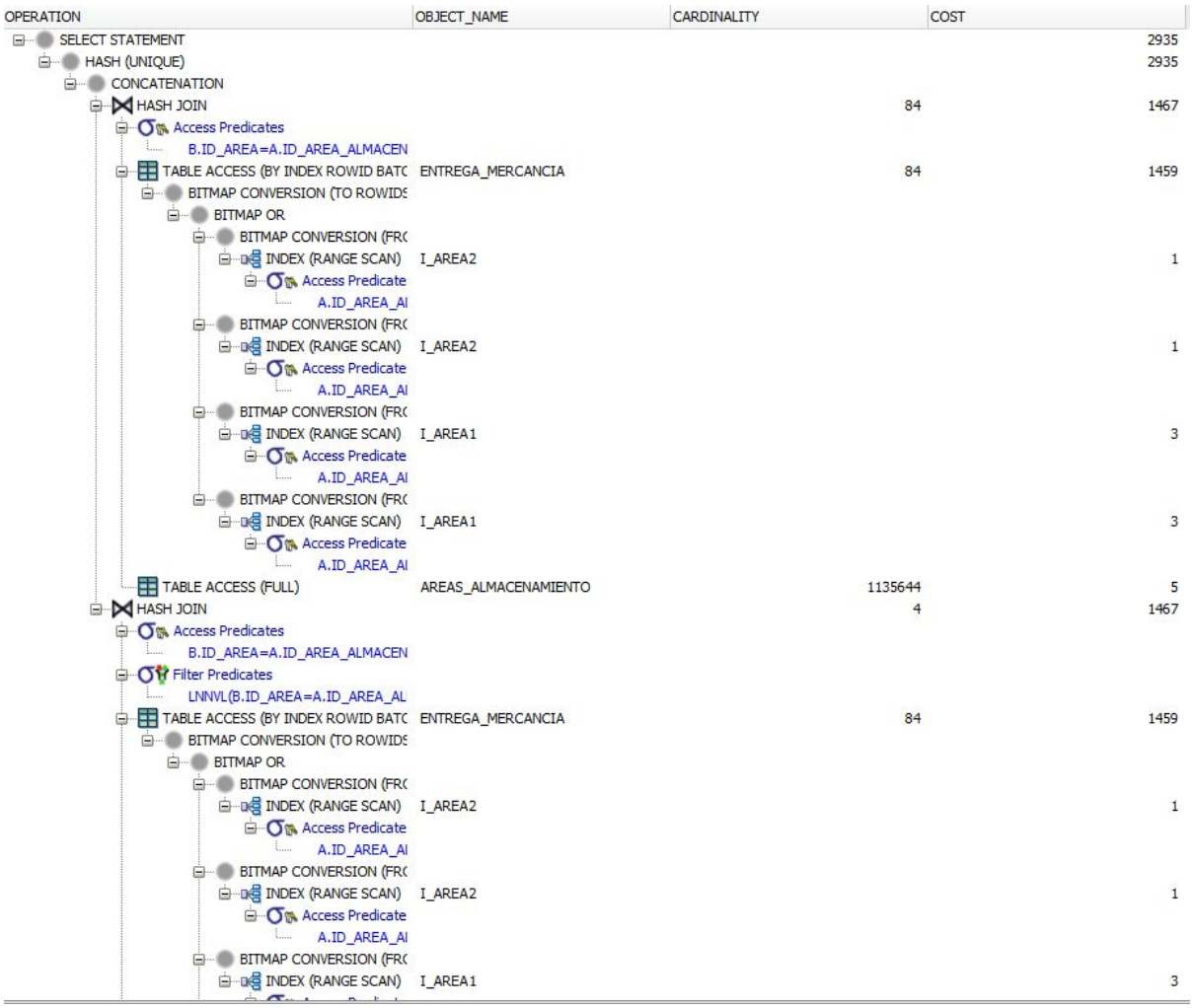
Salida de Script | Resultado de la Consulta | Explicación del Plan

SQL | 0,046 segundos

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT			4196
HASH (UNIQUE)			4196
CONCATENATION			
HASH JOIN		84	2097
Access Predicates			
B.ID_AREA=A.ID_AREA_ALMACENAMIENTO_2			
TABLE ACCESS (FULL)	ENTREGA_MERCANCIA	84	2089
Filter Predicates			
OR			
A.ID_AREA_ALMACENAMIENTO_2=1			
A.ID_AREA_ALMACENAMIENTO_2=153201			
A.ID_AREA_ALMACENAMIENTO=1			
A.ID_AREA_ALMACENAMIENTO=153201			
TABLE ACCESS (FULL)	AREAS_ALMACENAMIENTO	1135644	5
HASH JOIN		4	2097
Access Predicates			
B.ID_AREA=A.ID_AREA_ALMACENAMIENTO			
Filter Predicates			
LNNVL(B.ID_AREA=A.ID_AREA_ALMACENAMIENTO_2)			
TABLE ACCESS (FULL)	ENTREGA_MERCANCIA	84	2089
Filter Predicates			
OR			
A.ID_AREA_ALMACENAMIENTO_2=1			
A.ID_AREA_ALMACENAMIENTO_2=153201			
A.ID_AREA_ALMACENAMIENTO=1			
A.ID_AREA_ALMACENAMIENTO=153201			
TABLE ACCESS (FULL)	AREAS_ALMACENAMIENTO	1135644	5

Sin índices el optimizador realiza un hash para obtener valores únicos, de acuerdo al DISTINCT de la consulta. Este se hace entre la concatenación de dos hash join, de acuerdo a la condición del join. En el primer hash join se evalúa la condición en el que el id del área de almacenamiento sea igual al campo ID_AREA_ALMACENAMIENTO_2. En el segundo hash join se evalúa que este sea igual al campo ID_AREA_ALMACENAMIENTO. En ambos hash join se hace un acceso completo a las tablas Entrega_Mercancia y Areas_Almacenamiento. El costo es de 4196, con una cardinalidad aproximada de 88. El tiempo es de aproximadamente 0.05 segundos, que es muy poco.

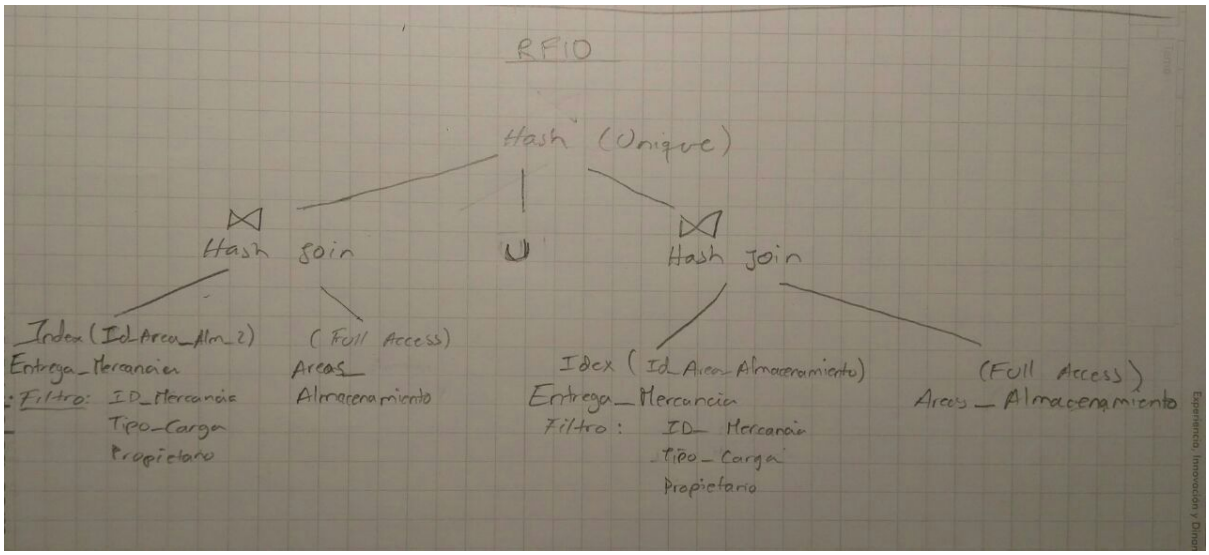
Con dos índices (Id_Area_Almacenamiento y Id_Area_Almacenamiento_2)



OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT			2935
HASH (UNIQUE)			2935
CONCATENATION			
HASH JOIN		84	1467
Access Predicates			
B.ID_AREA=A.ID_AREA_ALMACEN			
TABLE ACCESS (BY INDEX ROWID BATC	ENTREGA_MERCANCIA	84	1459
BITMAP CONVERSION (TO ROWIDS			
BITMAP OR			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA2		1
Access Predicate			
A.ID_AREA_AI			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA2		1
Access Predicate			
A.ID_AREA_AI			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA1		3
Access Predicate			
A.ID_AREA_AI			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA1		3
Access Predicate			
A.ID_AREA_AI			
TABLE ACCESS (FULL)	AREAS_ALMACENAMIENTO	1135644	5
HASH JOIN		4	1467
Access Predicates			
B.ID_AREA=A.ID_AREA_ALMACEN			
Filter Predicates			
LNNVL(B.ID_AREA=A.ID_AREA_AL			
TABLE ACCESS (BY INDEX ROWID BATC	ENTREGA_MERCANCIA	84	1459
BITMAP CONVERSION (TO ROWIDS			
BITMAP OR			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA2		1
Access Predicate			
A.ID_AREA_AI			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA2		1
Access Predicate			
A.ID_AREA_AI			
BITMAP CONVERSION (FRC			
INDEX (RANGE SCAN)	I_AREA1		3

Debido a que la búsqueda se hace con un OR, se consideró como mejor opción crear un índice para cada campo y que el optimizador usara ambos índices en momentos diferentes de la consulta. Aunque se sigue haciendo un acceso completo a la tabla de Mercancias, para los parámetros de la tabla Entrega_Mercancia se usan los índices creados, accediendo por el ROWID. Esto disminuye el costo a 2935, con la misma cardinalidad. Esta forma de manejar la consulta es recomendada pues los índices, aunque representan una cantidad de espacio en disco considerable, no cambian mucho (ejemplo: creación de nuevas áreas de almacenamiento) por lo que se hace fácil su mantenimiento.

Plan propuesto antes de ejecución sql (RF10)



Comparación planes de ejecución

En este caso, Oracle realizó un procedimiento similar al planteado por nosotros, solo que para el acceso a los índices usa una conversión a bitmap para realizar la operación OR.

Construcción de la aplicación y análisis de resultados

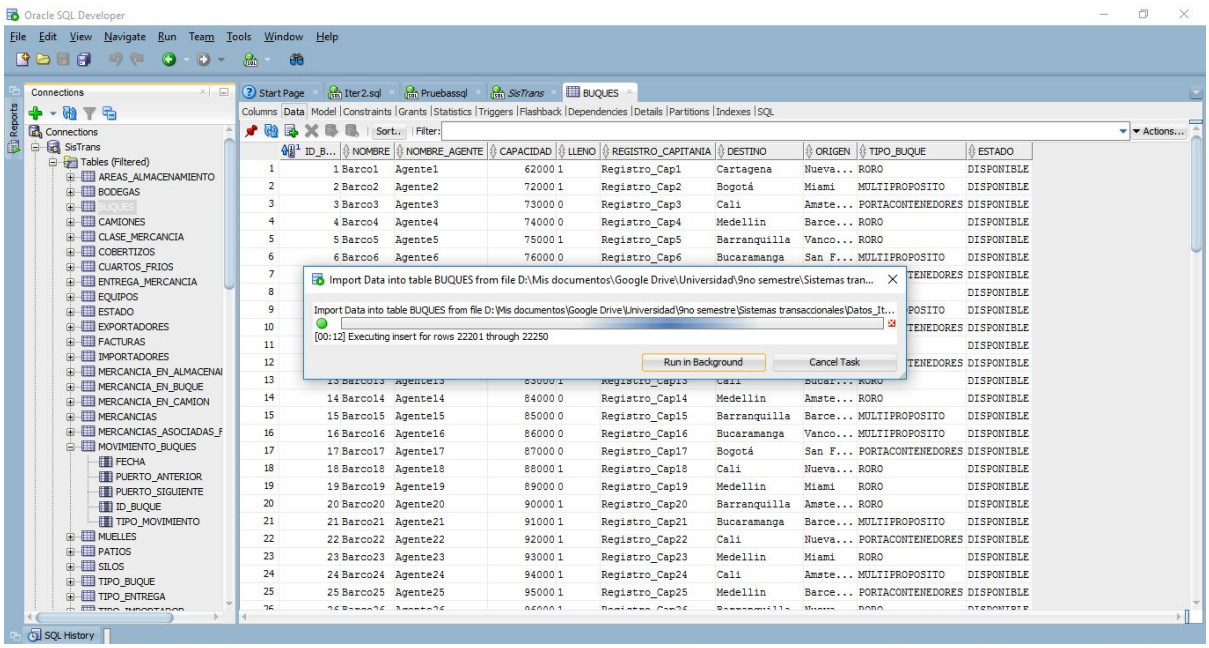
Carga de datos

Para la carga de datos se crearon archivos en Excel que se llenaron usando diferentes códigos escritos Visual Basic for Applications. Para poder tener una cantidad de datos que llenaran la memoria principal al realizarse las consultas, se crearon más de 1 millón de datos nuevos en las tablas involucradas en los requerimientos funcionales de consulta de esta iteración. A continuación se muestra uno de los códigos que utilizamos para poblar el Excel.

```
Sub datos()
    Dim i As Long
    For i = 1 To 1000000
        Cells(i, 1) = i + 100
        Cells(i, 2) = "Barco" & i Mod 80
        Cells(i, 3) = "Agente" & i Mod 80
        Cells(i, 4) = Int((10000 - 100 + 1) * Rnd() + 100)
        Cells(i, 5) = Int((2) * Rnd())
        Cells(i, 6) = "Registro" & i Mod 80
        Cells(i, 7) = "Ciudad" & Int((100 - 1 + 1) * Rnd() + 1)
        Cells(i, 8) = "Ciudad" & Int((100 - 1 + 1) * Rnd() + 1)
        Dim x As Double
        x = Rnd()
        If x > 0.67 Then
            Cells(i, 9) = "RORO"
```

```
        ElseIf x > 0.33 Then
            Cells(i, 9) = "MULTIPROPOSITO"
        Else
            Cells(i, 9) = "PORTACONTENEDORES"
        End If
        Cells(i, 10) = "DISPONIBLE"
    Next i
End Sub
```

Luego de generar los datos, se poblaron las tablas usando SQL Developer y su función de Import Data. En la siguiente imagen se muestra uno de los procesos de carga.



Análisis del proceso de optimización y modelo de ejecución de consultas

La diferencia radica en la eficiencia de la operación, ya que al realizarla en una aplicación se tendría que hacer un select de todas las tablas a consultar e ir subiendo los datos hasta completar la capacidad disponible, luego recorrer los registros y buscar la información necesaria usando instrucciones de control como if, while, etc para hacer filtros, esto indica que habría una gran cantidad de datos y en realidad sería como si no existieran los índices ni llaves primarias ya que tocaría recorrer toda la tabla, además el plan de ejecución podría no ser el más óptimo. Por otro lado, Oracle dispone desde un principio de trayectorias de acceso a la base de datos lo que agiliza la lectura de los datos, por esta razón se puede hacer uso de índices, también el manejador cuenta con un optimizador que se encarga de elegir el mejor plan de ejecución dependiendo de la consulta y el índice elegido.