

Sustentación de índices y análisis de datos Iteración 4

1. Con respecto al modelo anterior, se agregó el elemento de realizado en las funciones, así como se pulió un poco los nombres de los atributos que a veces causaban errores. Adicionalmente, para el requerimiento funcional 11 (el abonamiento) se debe de agregar una tabla completamente nueva de abonamientos, y su respectiva tabla de unión con las boletas. De ésta manera se manejan los abonamientos.

2.

Análisis de índices pre definidos por Oracle.

En nuestra base de datos, Oracle ya ha creado índices pre definidos en lo que él considera que son las consultas más comunes. Todos los índices que nos muestra Oracle son de IDs. **Aquí mostraré todos los índices que encontramos.**

INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124009	ISIS2304B161710	FESTIVALES	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124012	ISIS2304B161710	ROLES	CODROL		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124015	ISIS2304B161710	GENEROS	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124018	ISIS2304B161710	ABONAMIENTOS	IDABONO		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124026	ISIS2304B161710	SITIOS	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124033	ISIS2304B161710	ESPECTACULOS	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124038	ISIS2304B161710	USUARIOS	IDENTIFICACION		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124058	ISIS2304B161710	COMPANIASTEATRO	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124063	ISIS2304B161710	COMPANIACOOPERA	IDCOOPERACION		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124071	ISIS2304B161710	FUNCIONES	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124084	ISIS2304B161710	LOCALIDADES	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124090	ISIS2304B161710	BOLETAS	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124096	ISIS2304B161710	ABONAABOLETAS	IDABONACIONES		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124102	ISIS2304B161710	ESPECTACULOES	IDPREF		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124107	ISIS2304B161710	REQUERIMIENTOSTECNICOS	ID		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124115	ISIS2304B161710	ESPECTACULODEMANDA	IDDEMANDA		1 ASC
INDEX_OWNER	INDEX_NAME	TABLE_OWNER	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	DESCEND
ISIS2304B161710	SYS_C00124125	ISIS2304B161710	SITIO0FRECE	ID0FERTA		1 ASC

Como todos los índices que provee Oracle son por IDs, la búsqueda por IDs es muy efectiva, dado que están organizados ya como índice primario. Oracle los crea porque, dada la cantidad de registros, el DBMS crea uno predeterminado. Como no tenemos estadísticas, se crean únicamente de IDs. Sin embargo, en las consultas de la iteración, no sólo hay consultas por IDs.

Por ende, se deben crear nuevos índices.

Para los requerimientos funcionales del 3 al 7, no se requieren nuevos índices, ya que todos sus búsquedas son por IDs y éstos ya existen. Existen también búsquedas por rango de fechas, sin embargo, si se consulta en un rango superior a 6 meses, asumiendo una distribución uniforme de fechas, se estaría considerando más del 10% de las fechas, haciendo un índice para fechas en cualquier tabla innecesario, ya que costaría más usar el índice que no usarlo. Lo mismo aplica para los requerimientos funcionales 11 y 12, ya que si las localidades están distribuidas uniformemente, más del 10% de las localidades serán VIP, haciendo un índice inútil.

Por ende, se crearán 3 índices que nos permiten optimizar nuestros requerimientos funcionales de consulta.

- Un índice B+ sobre el nombre de los espectáculos. Será B+ por la cantidad de nombres que hay. Con un K grande, es muy eficiente. Ese será un índice secundario ya que sólo tendrá la dirección de los datos. Organizado por nombre, el requerimiento funcional 1 al buscar por nombre sólo deberá entrar al índice, encontrar el nombre que le piden, e ir a buscarlo.
 - Un índice hash sobre el ID del sitio ubicado en la tabla Localidades. Será hash ya que es una simple comparación la que se debe hacer, ahorrando mucho trabajo de compara 1 con 1 todo y yendo directamente a si está o no. Ese será un índice secundario ya que sólo tendrá la dirección de los datos. Organizado por ID en la tabla localidades, sólo se tendrá que hacer un hash para ver si existe, en vez de traer la tabla completa
- **** Como en estos requerimientos piden filtrado de datos, en realidad la cantidad de parámetros que existen no dan para generar índices. El único parámetro viable para generar índices es el del nombre de la compañía, que es el que más se usa, pero no hay muchas compañías de todas maneras. Si se genera otro índice, el costo de inserción de funciones se incrementa mucho, y no siempre se usarían los otros índices, porque esas consultas no son frecuentes. Se prefiere usar el mismo índice que se declaró antes.
- Un índice Hash sobre los ID Foráneos del espectáculo ubicados sobre la tabla Funciones. Será hash ya que es una simple comparación la que se debe hacer, ahorrando mucho trabajo de compara 1 con 1 todo y yendo directamente a si está o no. Ese será un índice secundario ya que sólo tendrá la dirección de los datos.

Prueba Requerimiento Funcional 1:

SQL:

```
SELECT FUNCIONES.ID, FUNCIONES.FECHAFUNCION, SITIOS.NOMBRE FROM
FUNCIONES, ESPECTACULOS, SITIOS
WHERE ESPECTACULOS.NOMBRE = 'espectaculo36'
AND ESPECTACULOS.ID = FUNCIONES.IDESPECTACULO
AND FUNCIONES.IDSITIO = SITIOS.ID
ORDER BY FUNCIONES.FECHAFUNCION;
```

SIN ÍNDICE:

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		10	2232
SORT (ORDER BY)		10	2232
HASH JOIN		10	2231
Access Predicates			
AND			
ESPECTACULOS.ID=FUNCIONES.IDESPECTACULO			
FUNCIONES.IDSITIO=SITIOS.ID			
MERGE JOIN (CARTESIAN)		200	1096
TABLE ACCESS (FULL)	ESPECTACULOS	1	1093
Filter Predicates			
BUFFER (SORT)		200	3
TABLE ACCESS (FULL)	SITIOS	200	3
TABLE ACCESS (FULL)	FUNCIONES	1000000	1132

Análisis:

Como vemos, el único índice que tenemos es el default que nos da Oracle. La consulta inicia haciendo acceso completo en sobre funciones y sitios, de los cuales filtra las funciones de los espectáculos cuyo nombre sea espectáculo36. Intenté haciendo un índice sobre ID para ver si alteraba algo, pero como vemos el proceso de HASH JOIN sólo cuesta 1, significando que Oracle ya está usando su índice predeterminado.

CON ÍNDICE:

SQL:

```
CREATE INDEX nombre_espectaculo ON ESPECTACULOS(NOMBRE);
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		10	1143
SORT (ORDER BY)		10	1142
HASH JOIN			
Access Predicates			
AND			
ESPECTACULOS.ID=FUNCIONES.IDESPECTACULO			
FUNCIONES.IDSITIO=SITIOS.ID			
MERGE JOIN (CARTESIAN)		200	7
TABLE ACCESS (BY INDEX ROWID)	ESPECTACULOS	1	4
INDEX (RANGE SCAN)	NOMBRE_ESPECTACULO	1	3
Access Predicates			
ESPECTACULOS.NOMBRE			
BUFFER (SORT)		200	3
TABLE ACCESS (FULL)	SITIOS	200	3
TABLE ACCESS (FULL)	FUNCIONES	1000000	1132

Análisis:

Como vemos, en el acceso a la tabla de espectáculos se está ahorrando casi 1000 de costo, una reducción sustancial comparado con el plan 1, sin índices. Ya no hace un acceso completo, si no que va directo a los nombres que le conciernen.

Prueba requerimiento funcional 2:

SQL:

```
SELECT ESPECTACULOS.NOMBRE AS NOMBRE_ESPECTACULO,
FUNCIONES.FECHAFUNCION,
LOCALIDADES.NOMBRE AS NOMBRE_LOCALIDAD, LOCALIDADES.PRECIO,
LOCALIDADES.CAPACIDAD AS CUPOS
FROM SITIOS, LOCALIDADES, ESPECTACULOS, FUNCIONES
WHERE ESPECTACULOS.ID = FUNCIONES.IDESPECTACULO
AND FUNCIONES.IDSITIO = SITIOS.ID
AND SITIOS.ID = LOCALIDADES.PERTENECEA
AND SITIOS.NOMBRE = 'Sitio55'
ORDER BY NOMBRE_ESPECTACULO;
SIN ÍNDICE:
```

OPERATION	OBJECT_NAME	CARDINALITY	COST	
SELECT STATEMENT			15766	19965
SORT (ORDER BY)			15766	19965
HASH JOIN			15766	19649
Access Predicates				
SITIOS.ID=LOCALIDADES.PERTENE				
HASH JOIN		5000		2231
Access Predicates				
ESPECTACULOS.ID=FUNCIONE				
HASH JOIN		5000		1138
Access Predicates				
FUNCIONES.IDSITIO=SITIO				
TABLE ACCESS (FULL)	SITIOS	1		3
Filter Predicates				
SITIOS.NOMBRE='Sitio'				
TABLE ACCESS (FULL)	FUNCIONES	1000000		1132
TABLE ACCESS (FULL)	ESPECTACULOS	300000		1093
TABLE ACCESS (FULL)	LOCALIDADES	3000000		17410

Análisis:

Como en la base de datos tenemos muchísimos registros de Localidades y funciones, y como cada función tiene un sitio y un sitio muchas localidades, la repetición de datos fue muchísima. Por ende, es una consulta grande, que debe buscar sobre todas las localidades. Esto es lo que más tiempo consume.

CON ÍNDICE:

SQL:

```
CREATE INDEX foreignllavesitios ON LOCALIDADES(pertenecea);
```

OPERATION	OBJECT_NAME	CARDINALITY	COST	
SELECT STATEMENT			15766	2554
SORT (ORDER BY)			15766	2237
HASH JOIN			15766	2237
Access Predicates				
NESTED LOOPS				
NESTED LOOPS				
STATISTICS COLLECTOR				
HASH JOIN		15766		1144
Access Predicates				
HASH JOIN		3		9
Access Predicates				
NESTED LOOPS		3		9
NESTED LOOPS		3		9
STATISTICS COLLECTOR				
TABLE ACCESS (FULL) SITIOS		1		3
INDEX (RANGE SCAN)	FOREIGNLLAVESITIOS	3		2
TABLE ACCESS (BY INDEX ROW)	LOCALIDADES	3		6
TABLE ACCESS (FULL)	LOCALIDADES	3		6
TABLE ACCESS (FULL)	FUNCIONES	1000000		1132
INDEX (UNIQUE SCAN)	SYS_C00124033			
TABLE ACCESS (BY INDEX ROWID)	ESPECTACULOS	1		1093
TABLE ACCESS (FULL)	ESPECTACULOS	300000		1093

Análisis:

Como es evidente, el costo de ejecución de la consulta se reduce casi en 90%. El DBMS cambia por completo la estructura del Join incluso, ya que con el índice se ahorra muchísimo tiempo comparando TODAS las localidades con TODOS los Sitios, solamente tiene que hacer una función hashing y verificar si está o no. Nos ahorramos un acceso completo a la tabla localidades, que ahora sale en gris.

Prueba requerimiento funcional 8:**SQL:**

```
SELECT SITIOS.ID FROM COMPANIASTEATRO, COMPANIACOOPERA,
ESPECTACULOS, FUNCIONES, SITIOS
WHERE COMPANIASTEATRO.NOMBRE = 'compania6'
AND COMPANIACOOPERA.IDCOMPANIA = COMPANIASTEATRO.ID
AND COMPANIACOOPERA.IDESPECTACULO = ESPECTACULOS.ID
AND ESPECTACULOS.ID = FUNCIONES.IDESPECTACULO
AND FUNCIONES.IDSITIO = SITIOS.ID;
```

```
SELECT COUNT(BOLETAS.ID) FROM SITIOS, LOCALIDADES, BOLETAS
WHERE SITIOS.ID = 25
AND LOCALIDADES.PERTENECEA = SITIOS.ID
AND LOCALIDADES.ID = BOLETAS.ID;
SIN ÍNDICE:
```

OPERATION	OBJECT_NAME	CARDINALITY	COST	
SELECT STATEMENT			136	1140
HASH JOIN			136	1140
Access Predicates				
AND				
COMPANIACOOPERA.IDCOMPANIA=COMPANIAS				
FUNCIONES.IDESPECTACULO=COMPANIACOOPERA				
TABLE ACCESS (FULL)	COMPANIACOOPERA	110		3
MERGE JOIN (CARTESIAN)			1000000	1135
TABLE ACCESS (FULL)	COMPANIASTEATRO	1		3
Filter Predicates				
COMPANIASTEATRO.NOMBRE='compania6'				
BUFFER (SORT)		1000000		1132
TABLE ACCESS (FULL)	FUNCIONES	1000000		1132

OPERATION	OBJECT_NAME	CARDINALITY	COST	
SELECT STATEMENT			1	6
SORT (AGGREGATE)			1	
NESTED LOOPS			3	6
NESTED LOOPS			3	6
INDEX (UNIQUE SCAN)	SYS_C00124026		1	0
Access Predicates				
SITIOS.ID=25				
TABLE ACCESS (BY INDEX ROWID BATCHED)	LOCALIDADES	3		6
INDEX (RANGE SCAN)	FOREIGNLLAVESITIOS	3		2
Access Predicates				
LOCALIDADES.PERTENECEA=25				
INDEX (UNIQUE SCAN)	SYS_C00124366		1	0
Access Predicates				
LOCALIDADES.ID=BOLETAS.ID				

Análisis:

En el requerimiento funcional 8, se debe buscar todos los sitios en dónde la compañía vendió boletas, para así generar luego las estadísticas ejecutando un loop contando todas las boletas y multiplicándolas por el precio de su localidad. Por ende, se ejecuta 1 vez el SQL superior, que es la foto de arriba, y un loop dependiendo de la cantidad de resultados en el sql inferior, que es la foto de abajo. Como vemos, la consulta está haciendo uso de un índice que

declaramos previamente, ya que en el loop se están haciendo parte de la misma consulta del RF2. Sin embargo, lo que más consume tiempo en ésta consulta es la búsqueda de los espectáculos y sitios que corresponden a las funciones, por lo que se hará un índice ahí.

CON ÍNDICE:

SQL:

CREATE INDEX foreignllavefuncesp **ON** FUNCIONES(IDESPECTACULO);

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT			136
HASH JOIN			174
Access Predicates			136
FUNCIONES.IDESPECTACULO=COMPANIACOOPERA.			174
NESTED LOOPS		136	174
NESTED LOOPS		140	174
STATISTICS COLLECTOR			
HASH JOIN		14	6
Access Predicates			
COMPANIACOOPERA.IDCOMPANIA=(
TABLE ACCESS (FULL)	COMPANIASTEATRO	1	3
Filter Predicates			
COMPANIASTEATRO.NOMBRE='c'			
TABLE ACCESS (FULL)	COMPANIACOOPERA	110	3
INDEX (RANGE SCAN)	FOREIGNLLAVEFUNCESP	10	2
Access Predicates			
FUNCIONES.IDESPECTACULO=COMPANI/			
TABLE ACCESS (BY INDEX ROWID)	FUNCIONES	10	12
TABLE ACCESS (FULL)	FUNCIONES	10	12

Análisis:

Si bien con éste índice el loop inferior no se altera, pues éste ya usa un índice, la consulta inicial para obtener la lista de datos se altera drásticamente, reduciendo en 85% su tiempo de ejecución. Esto se debe a que ya no debe hacer comparaciones 1 con 1 con cada elemento de las tablas funciones y espectáculos, si no que automáticamente obtiene aquellos que concuerdan mediante su índice, evitándonos 2 accesos completos a tablas en disco.

Prueba requerimiento funcional 9 y 10:

SQL:

```
SELECT USUARIOS.NOMBRE, USUARIOS.EMAIL FROM COMPANIASTEATRO,
COMPANIACOOPERA, ESPECTACULOS, FUNCIONES, SITIOS, LOCALIDADES,
BOLETAS, USUARIOS
WHERE COMPANIASTEATRO.NOMBRE = 'compania3'
AND COMPANIASTEATRO.ID = COMPANIACOOPERA.IDCOMPANIA
AND COMPANIACOOPERA.IDESPECTACULO = ESPECTACULOS.ID
AND ESPECTACULOS.ID = FUNCIONES.IDESPECTACULO
AND FUNCIONES.IDSITIO = SITIOS.ID
AND SITIOS.ID = LOCALIDADES.PERTENECEA
AND LOCALIDADES.ID = BOLETAS.IDLOCALIDAD
AND BOLETAS.IDCLIENTE = USUARIOS.IDENTIFICACION;
```

SIN ÍNDICE:

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		551	4089
HASH JOIN		551	4089
Access Predicates			
NESTED LOOPS		551	4089
NESTED LOOPS		551	4089
STATISTICS COLLECTOR			
HASH JOIN		551	2987
Access Predicates			
HASH JOIN		430	1956
Access Predicates			
NESTED LOOPS		430	1956
NESTED LOOPS		430	1956
STATISTICS COLLECTOR			
HASH JOIN		136	1140
Access Predicates			
HASH JOIN		14	6
Access Predicates			
TABLE ACCESS (FULL) COMPANIASTEATRO		1	3
TABLE ACCESS (FULL) COMPANIACOOPERA		110	3
TABLE ACCESS (FULL) FUNCIONES		1000000	1132
INDEX (RANGE SCAN) FOREIGNLLAVESITIOS		3	2
TABLE ACCESS (BY INDEX ROWID) LOCALIDADES		3	6
TABLE ACCESS (FULL) LOCALIDADES		3	6
TABLE ACCESS (FULL) BOLETAS		815231	1028
INDEX (UNIQUE SCAN) SYS_C00124038		1	1
TABLE ACCESS (BY INDEX ROWID) USUARIOS		1	2
TABLE ACCESS (FULL) USUARIOS		1	2

Análisis:

Este requerimiento pasa por los mismos pasos que los requerimientos anteriores, re usando sus índices. Si los quitamos, observamos el resultado de arriba. Estamos haciendo 4 accesos completos a tablas, lo que está consumiendo un tiempo razonable. Por ende, el mismo índice anterior mejora la situación.

CON ÍNDICE:

SQL:

CREATE INDEX foreignllavefuncesp **ON** FUNCIONES(IDESPECTACULO);

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		551	3123
HASH JOIN		551	3123
Access Predicates			
NESTED LOOPS		551	3123
NESTED LOOPS		551	3123
STATISTICS COLLECTOR			
HASH JOIN		551	2021
Access Predicates			
HASH JOIN		430	990
Access Predicates			
NESTED LOOPS		430	990
NESTED LOOPS		430	990
STATISTICS COLLECTOR			
HASH JOIN		136	174
Access Predicates			
NESTED LOOPS		136	174
STATISTICS COLLECTOR			
HASH JOIN		14	6
Access Predicates			
TABLE ACCESS (FULL) COMPANIASTEATRO		1	3
TABLE ACCESS (FULL) COMPANIACOOPERA		110	3
TABLE ACCESS (BY INDEX ROWID) FUNCIONES		10	12
TABLE ACCESS (FULL) FUNCIONES		10	12
INDEX (RANGE SCAN) FOREIGNLLAVESITIOS		3	2
TABLE ACCESS (BY INDEX ROWID) LOCALIDADES		3	6
TABLE ACCESS (FULL) LOCALIDADES		3	6
TABLE ACCESS (FULL) BOLETAS		815231	1028
INDEX (UNIQUE SCAN) SYS_C00124038		1	1
TABLE ACCESS (BY INDEX ROWID) USUARIOS		1	2

Análisis:

La mayoría de tiempo se está yendo en el acceso completo sobre la tabla boletas, ya que es en donde más hay registros. En ésta tabla, probé con muchos índices pero todos ocupan más del 10% de la memoria, por lo que el DBMS decide no usarlos ya que gastaría más tiempo. Sin embargo, con el índice anterior, se ahorra casi un 25% del costo, lo cual es un aumento sustancial del tiempo de ejecución.

Pruebas DEMAS requerimientos funcionales:**RF3:****SQL:**

```
SELECT COUNT(BOLETAS.ID) FROM FUNCIONES, SITIOS, LOCALIDADES,
BOLETAS
WHERE FUNCIONES.ID = 56
AND FUNCIONES.IDSITIO = SITIOS.ID
AND LOCALIDADES.PERTENECEA = SITIOS.ID
AND LOCALIDADES.ID = BOLETAS.ID;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	9
SORT (AGGREGATE)		1	
NESTED LOOPS		3	9
NESTED LOOPS		3	9
TABLE ACCESS (BY INDEX ROWID)	FUNCIONES	1	3
INDEX (UNIQUE SCAN)	SYS_C00124071	1	2
Access Predicates			
FUNCIONES.ID=56			
TABLE ACCESS (BY INDEX ROWID BATCHED)	LOCALIDADES	3	6
INDEX (RANGE SCAN)	FOREIGNLLAVESITIOS	3	2
Access Predicates			
LOCALIDADES.PERTENECEA=FUNCIOI			
INDEX (UNIQUE SCAN)	SYS_C00124366	1	0
Access Predicates			
LOCALIDADES.ID=BOLETAS.ID			

RF4:**SQL:**

```
SELECT COUNT(BOLETAS.ID) FROM ESPECTACULOS, FUNCIONES, SITIOS,
LOCALIDADES, BOLETAS
WHERE ESPECTACULOS.NOMBRE = 'espectaculo16'
AND FUNCIONES.IDESPECTACULO = ESPECTACULOS.ID
AND FUNCIONES.IDSITIO = SITIOS.ID
AND LOCALIDADES.PERTENECEA = SITIOS.ID
AND LOCALIDADES.ID = BOLETAS.ID;
```


OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	76
SORT (AGGREGATE)		1	
NESTED LOOPS		31	76
HASH JOIN		31	76
Access Predicates			
NESTED LOOPS		31	76
STATISTICS COLLECTOR			
HASH JOIN		10	16
Access Predicates			
NESTED LOOPS		10	16
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWESPECTACULOS		1	4
INDEX (RANGE SCAN) NOMBRE_ESPECTACULO		1	3
TABLE ACCESS (BY INDEX ROWID FUNCIONES		10	12
INDEX (RANGE SCAN) FOREIGNLLAVEFUNCESP		10	2
TABLE ACCESS (FULL) FUNCIONES		10	12
TABLE ACCESS (BY INDEX ROWID BATCHED) LOCALIDADES		3	6
INDEX (RANGE SCAN) FOREIGNLLAVESITIOS		3	2
TABLE ACCESS (FULL) LOCALIDADES		3	6
INDEX (UNIQUE SCAN) SYS_C00124366		1	0

RF6:

Hay que repetir el query con cada ID de espectáculo, y luego organizarlo en el top.

SQL:

```
SELECT COUNT(BOLETAS.ID) FROM ESPECTACULOS, FUNCIONES, SITIOS,
LOCALIDADES, BOLETAS
WHERE ESPECTACULOS.ID = 1
AND FUNCIONES.IDESPECTACULO = ESPECTACULOS.ID
AND FUNCIONES.FECHAFUNCION BETWEEN '01/01/2011' AND '01/01/2013'
AND FUNCIONES.IDSITIO = SITIOS.ID
AND LOCALIDADES.PERTENECEA = SITIOS.ID
AND LOCALIDADES.ID = BOLETAS.ID;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	31
SORT (AGGREGATE)		1	
NESTED LOOPS		9	31
HASH JOIN		9	31
Access Predicates			
NESTED LOOPS		9	31
STATISTICS COLLECTOR			
TABLE ACCESS (BY INDEX ROWID BATCHED) FUNCIONES		3	13
Filter Predicates			
INDEX (RANGE SCAN) FOREIGNLLAVEFUNCESP		10	3
TABLE ACCESS (BY INDEX ROWID BATCHED) LOCALIDADES		3	6
INDEX (RANGE SCAN) FOREIGNLLAVESITIOS		3	2
TABLE ACCESS (FULL) LOCALIDADES		3	6
INDEX (UNIQUE SCAN) SYS_C00124366		1	0
Access Predicates			

RF7:

Probé con varios índices y no mejoró.

SQL:

```
SELECT FUNCIONES.ID, FUNCIONES.FECHAFUNCION FROM USUARIOS,
BOLETAS, LOCALIDADES, SITIOS, FUNCIONES
```

WHERE USUARIOS.IDENTIFICACION = 416301
 AND USUARIOS.IDENTIFICACION = BOLETAS.IDCLIENTE
 AND BOLETAS.IDLOCALIDAD = LOCALIDADES.ID
 AND SITIOS.ID = LOCALIDADES.PERTENECEA
 AND SITIOS.ID = FUNCIONES.IDSITIO;

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		44	2246
HASH JOIN		44	2246
Access Predicates			
FUNCIONES.IDSITIO=LOCALIDADES.PERTENECEA			
HASH JOIN		41	1111
Access Predicates			
BOLETAS.IDLOCALIDAD=LOCALIDADES.ID			
NESTED LOOPS		41	1111
NESTED LOOPS		41	1111
STATISTICS COLLECTOR			
TABLE ACCESS (FULL)	BOLETAS	41	1029
Filter Predicates			
BOLETAS.IDCLIENTE=416301			
INDEX (UNIQUE SCAN)	SYS_C00124084	1	1
Access Predicates			
BOLETAS.IDLOCALIDAD=LOCALIDAD			
TABLE ACCESS (BY INDEX ROWID)	LOCALIDADES	1	2
TABLE ACCESS (FULL)	LOCALIDADES	1	2
TABLE ACCESS (FULL)	FUNCIONES	1000000	1132