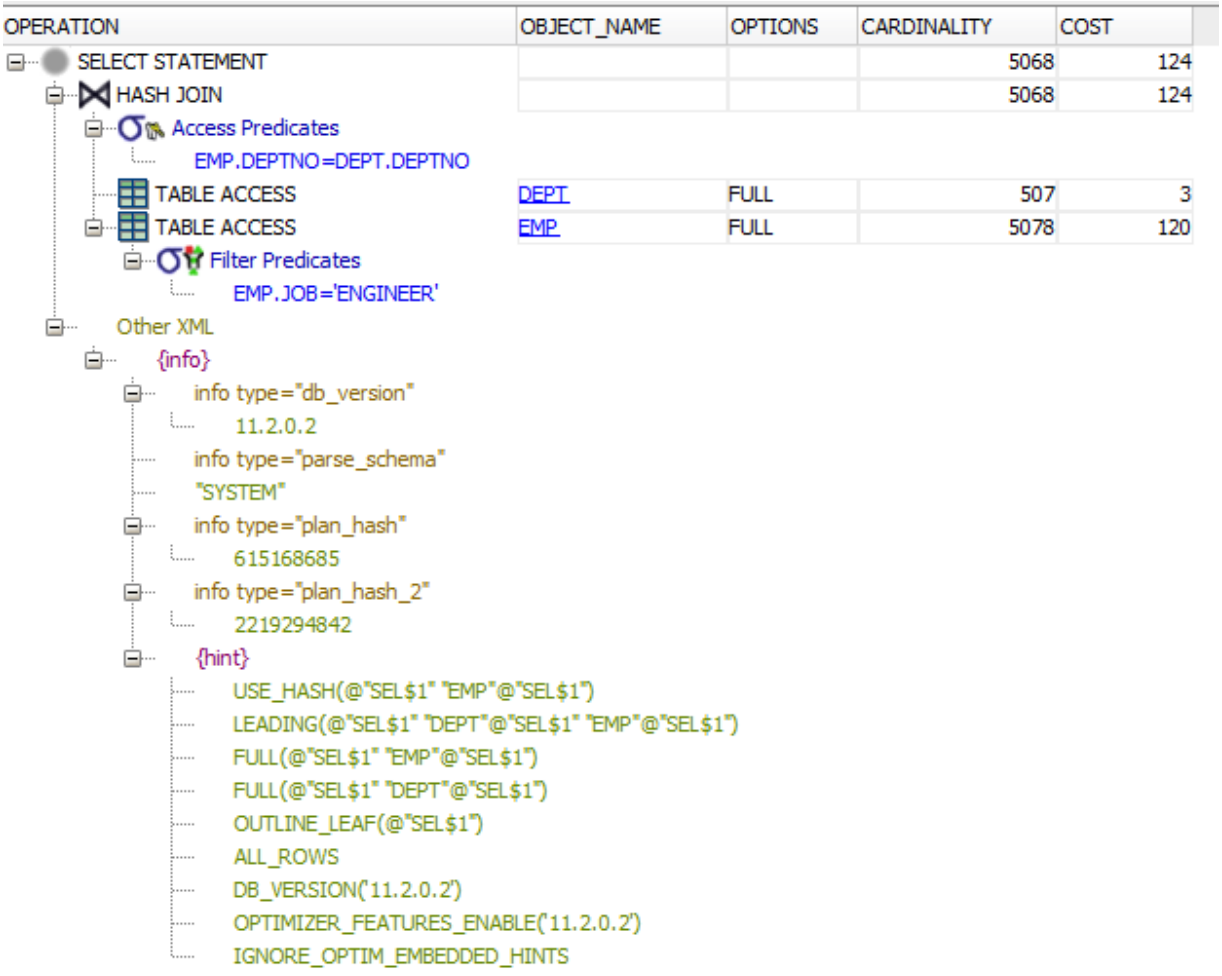
Esercitazione 5 – Ottimizzatore di Oracle

Query #1

```
SELECT /*+ FIRST_ROWS(n) */ *
FROM emp, dept
WHERE emp.deptno = dept.deptno AND emp.job = 'ENGINEER';
```



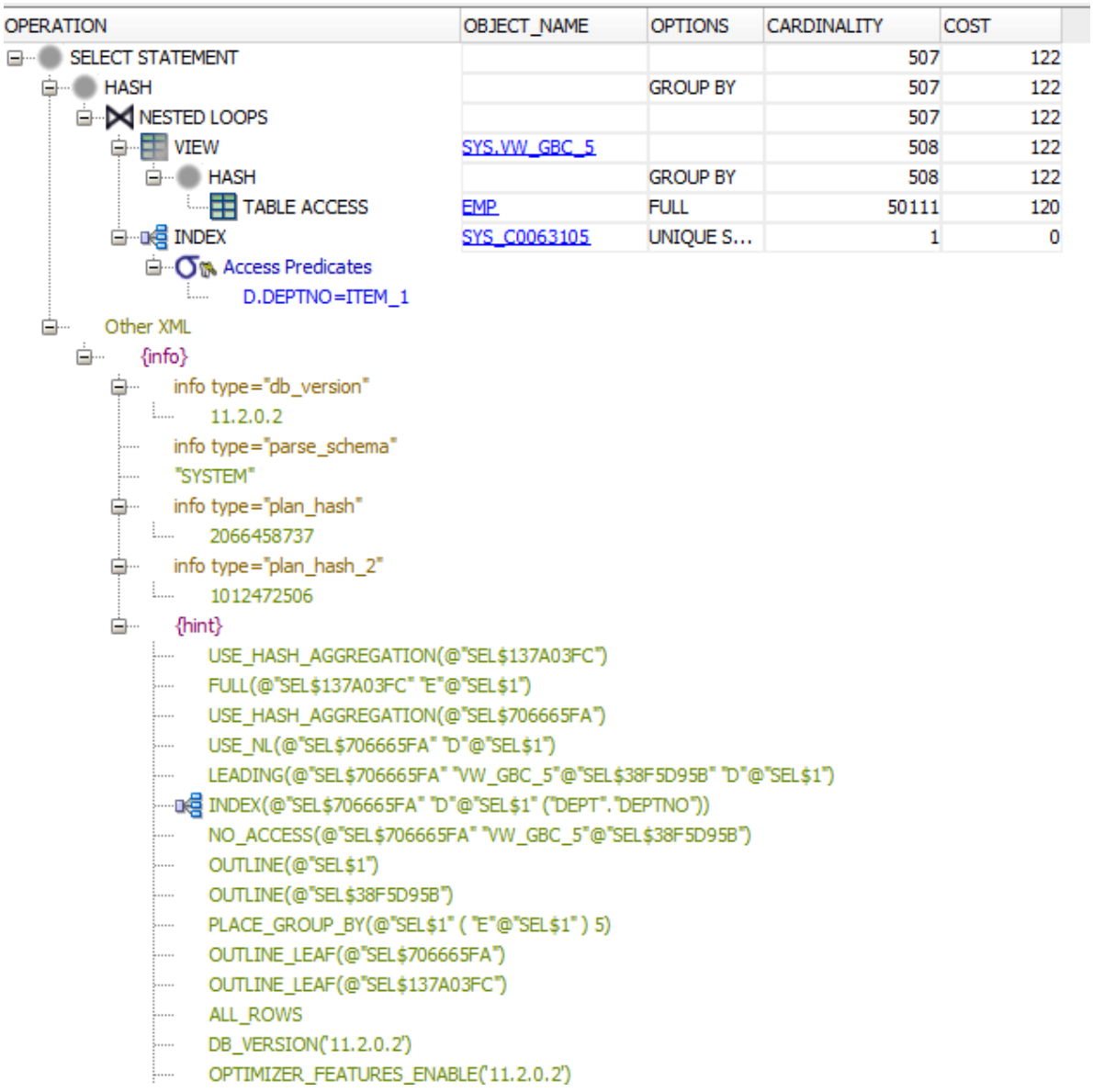
- Viene eseguita una **Hash Join** su DEPTNO, in seguito al **filtro** delle righe su attributo Job. Accesso alla tabella: full (= lettura di tutte le righe).
- Il **costo** rappresentato sulla colonna a destra è cumulativo. Questo incrementa infatti seguendo il percorso dalle foglie alla radice dell'albero del piano di esecuzione. Ad ogni operazione viene incrementato del costo di quest'ultima. In questo esempio le foglie dell'albero sono 2: i due table access su *dept* ed *emp*, con costo rispettivo 3 e 120. Salendo nell'albero i costi dei due accessi vengono sommati insieme al costo della Hash Join (3+120+1 = 124). La hash join ha infatti costo unitario.

Query #2

Confrontare i costi di hash join e nested loop usando l'hint USE / NO_USE_HASH

```
SELECT /*+ NO_USE_HASH(e d) */ d.deptno, AVG(e.sal)
FROM emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.deptno;
```

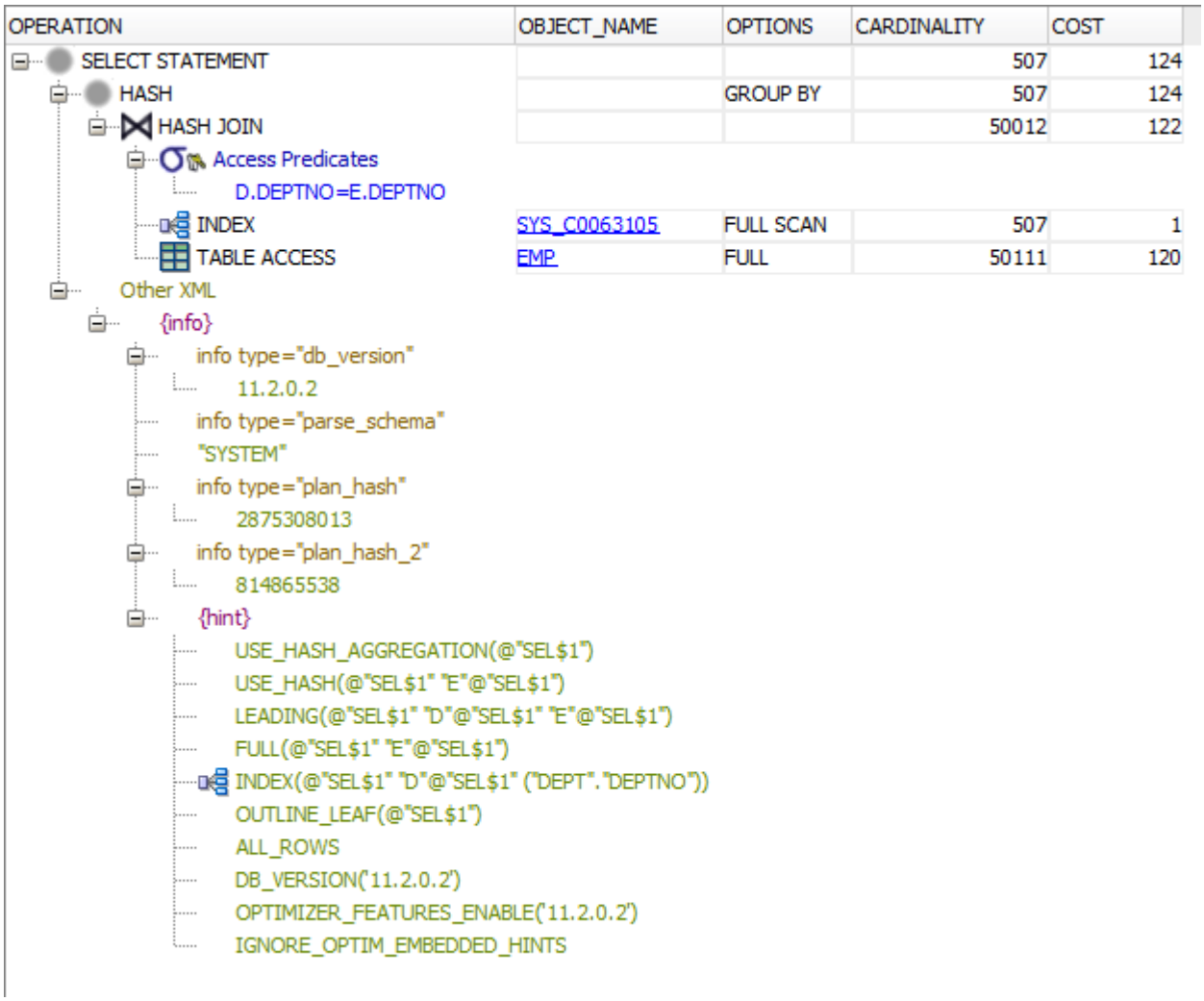
Con hint NO_USE_HASH:



- DEPTNO = ITEM_1: quando la Group By viene eseguita, si crea una **vista** in cui l'attributo chiave (DEPTNO) viene rinominato con ITEM_1. ITEM_1 diventa inoltre **l'indice** della vista per un accesso rapido ai gruppi.
- L'operazione di GROUP BY compare due volte. Quella più interna rappresenta **l'anticipo** della group by (anticipata prima della join). Quella più esterna rappresenta la Group By nella posizione iniziale, prima dell'anticipo. Infatti se si osserva il **costo cumulativo**, questo incrementa solo con la group by interna

(che svolge il lavoro) e non con quella esterna (che non deve più raggruppare in quanto i gruppi sono già stati creati da quella più interna).

Con hint USE_HASH:



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			507	124
HASH		GROUP BY	507	124
HASH JOIN			50012	122
Access Predicates				
D.DEPTNO=E.DEPTNO				
INDEX	SYS_C0063105	FULL SCAN	507	1
TABLE ACCESS	EMP	FULL	50111	120

- Qui viene utilizzata una **hash join**. La group by invece non viene anticipata. Nonostante la hash join, la sequenza di queste operazioni rimane meno efficiente rispetto al punto precedente (probabilmente a causa del mancato anticipo della GROUP BY).

Query #3

Disabilitare il metodo hash join mediante l'uso di hint (/*+ NO_USE_HASH(e d) */)

```
SELECT /*+ NO_USE_HASH(e d) */ ename, job, sal, dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND NOT EXISTS
    (SELECT * FROM salgrade WHERE e.sal = hisal);
```

SQL | 0,03 secondi

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			50012	530
HASH JOIN		RIGHT ANTI	50012	530
Access Predicates				
E.SAL=HISAL				
TABLE ACCESS	SALGRADE	FULL	999	3
MERGE JOIN			50012	527
SORT		JOIN	507	4
TABLE ACCESS	DEPT	FULL	507	3
SORT		JOIN	50111	523
Access Predicates				
E.DEPTNO=D.DEPTNO				
Filter Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	EMP	FULL	50111	120
Other XML				
{info}				
info type="db_version"				
11.2.0.2				
info type="parse_schema"				
"SYSTEM"				
info type="plan_hash"				
3726606473				

Nota. Per scrivere lo schema della query in algebra relazionale, l'operatore "NOT EXISTS" può essere riscritto con una **NOT IN**. Infatti:

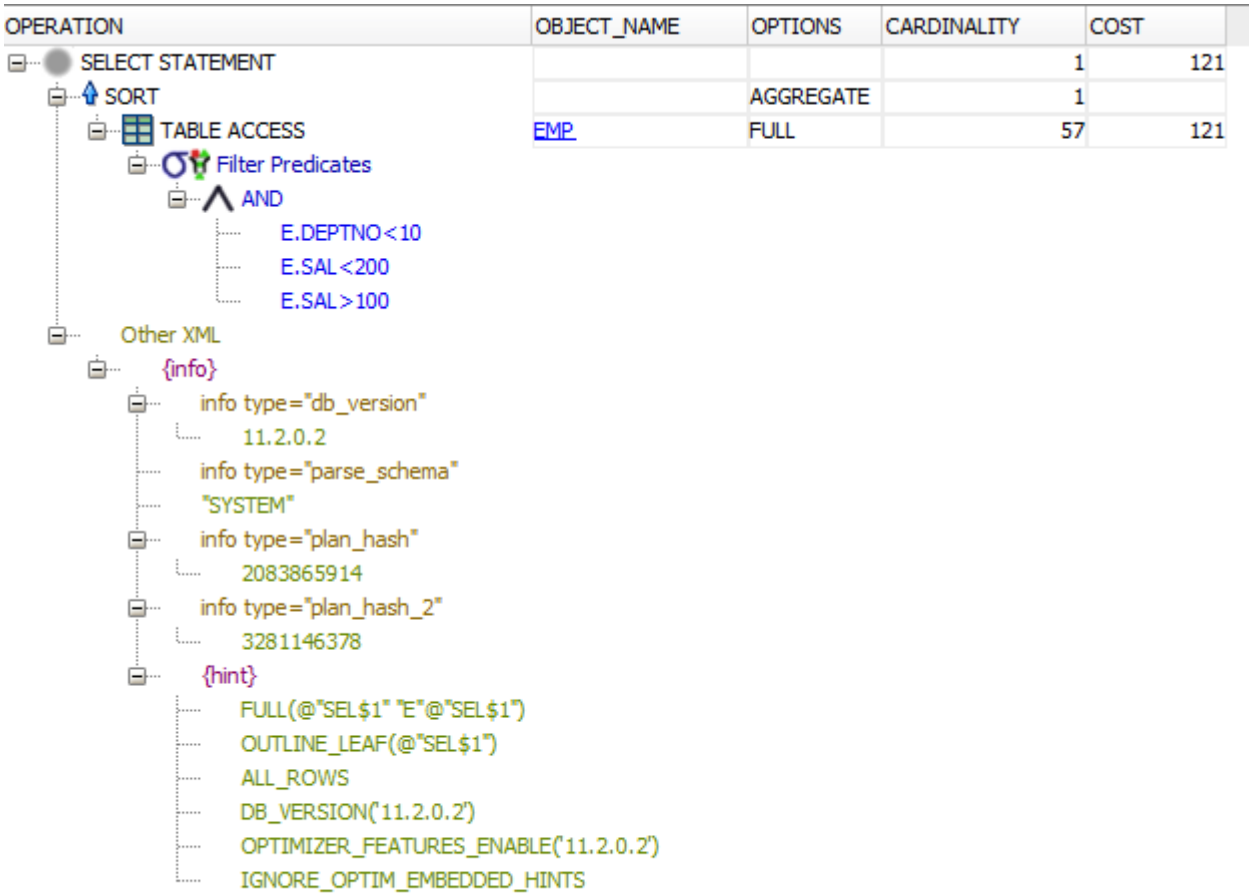
```
SELECT ename, job, sal, dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND e.sal NOT IN
    (SELECT hisal FROM salgrade);
```

Query #4

Si definiscano una o più strutture secondarie (indici) che permettano l'ottimizzazione della seguente query. Si analizzi con particolare attenzione il cambiamento nel piano di esecuzione creando due indici sugli attributi interessati dall'interrogazione.

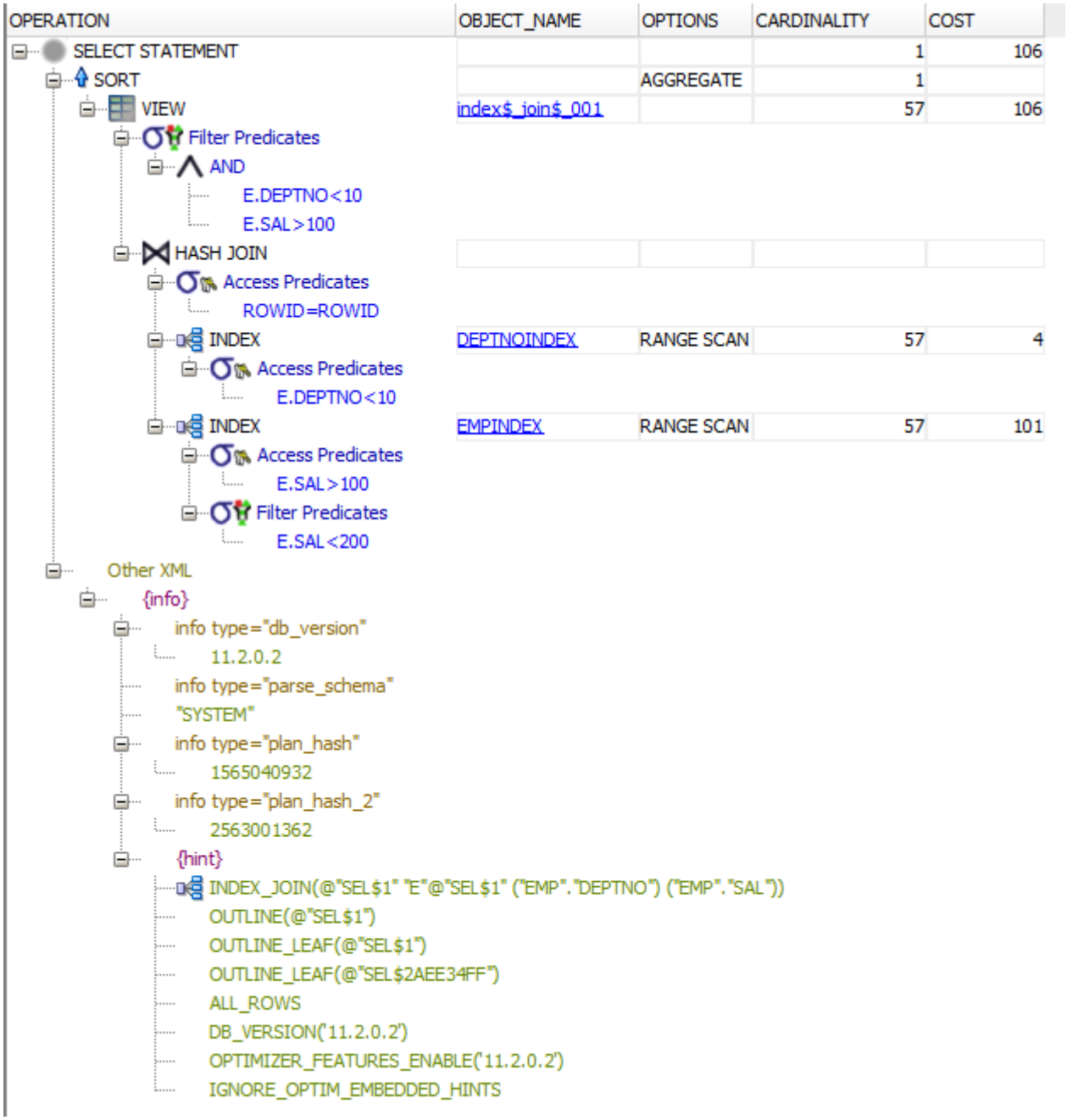
```
select avg(e.sal)
from emp e
where e.deptno < 10 and
e.sal > 100 and e.sal < 200;
```

Senza indici:



Con indici (secondari) su sal e deptno:

```
create index SalIndex on EMP(sal)
create index DepIndex on EMP(deptno)
```



- Gli indici vengono utilizzati per selezionare (**access predicates**) le righe in base a *Sal* e *Deptno*.
- Un access predicate indica, nel caso di un **B+Tree index**, che i dati vengono selezionati in base ad un certo attributo scendendo nella **gerarchia dell'albero** (**costo logaritmico**). Le **foglie** dell'albero raggruppano i dati in modo

"**grossolano**" (=una foglia può contenere molti valori per l'attributo indicizzato).

- Per questo motivo spesso ad un access predicate **segue un filter predicate**, in cui si analizzano i dati nella foglia dell'albero selezionata. Durante il filter predicate viene quindi eseguito un **filtro più fine** sui valori dell'attributo. Questa operazione **ha costo lineare** (tutti i dati nella foglia vengono analizzati).

- In questo esempio il B+Tree viene usato per selezionare Sal>100, poi la ricerca lineare viene usata per selezionare i dati con Sal<200.
- Successivamente viene utilizzato l'indice su DeptNo per selezionare i dipartimenti.
- La Hash Join serve per combinare (tramite row id) i risultati ottenuti filtrando con i due indici (intersezione delle righe che soddisfano le condizioni).

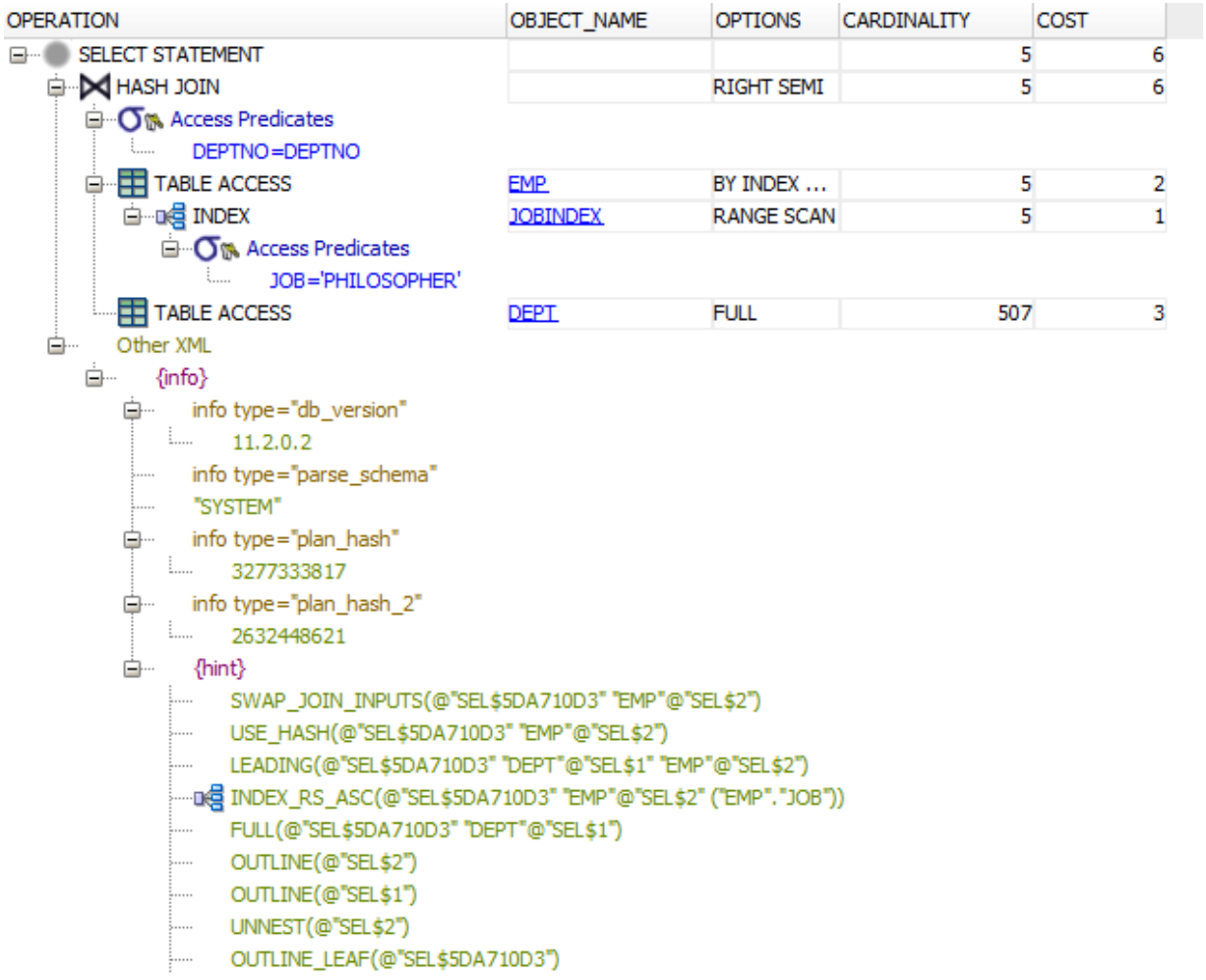
- I filter predicates esterni (deptno<10 e sal>100) vengono infine applicati perché su questi due attributi, nella parte interna della query, è stato eseguito solo un access predicate. Come spiegato in precedenza, un access predicate non è sufficientemente raffinato per selezionare gli esatti valori desiderati, per questo diventa necessario questo ulteriore filter predicate.

Query #5

Si definiscano una o più strutture secondarie (indici) che permettano l'ottimizzazione della seguente query:

```
select dname
from dept
where deptno in (select deptno
                 from emp
                 where job = 'PHILOSOPHER');
```

Indice su emp(job), dato che la query interna esegue un filtro su questo attributo. Un ulteriore indice su emp(deptno) non aiuterebbe la join, poiché prima della join viene eseguito un filtro su Job, che richiede l'indice su Job ed un access by row id.



- Il sistema esegue una lettura dell'indice su JOB per filtrare le righe di EMP.
- Successivamente esegue un access by row id (OPTIONS=BY INDEX) per accedere al contenuto degli altri attributi di EMP.
- Infine esegue una hash join con la tabella DEPT

Query #6

Si definiscano una o più strutture secondarie (indici) che permettano l'ottimizzazione della seguente query (rimuovere eventuali indici già esistenti per confrontare le performance della query con e senza indici):

```
select e1.ename, e1.empno, e1.sal, e2.ename, e2.empno, e2.sal
from emp e1, emp e2
where e1.ename <> e2.ename and e1.sal < e2.sal
and e1.job = 'PHILOSOPHER' and e2.job = 'ENGINEER';
```

Indice su emp(job), emp(name) e emp(sal):

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12664	125
MERGE JOIN			12664	125
SORT		JOIN	5	3
TABLE ACCESS	EMP	BY INDEX ...	5	2
INDEX	JOBINDEX	RANGE SCAN	5	1
Access Predicates				
E1.JOB='PHILOSOPHER'				
FILTER				
Filter Predicates				
E1.ENAME<>E2.ENAME				
SORT		JOIN	5078	122
Access Predicates				
E1.SAL<E2.SAL				
Filter Predicates				
E1.SAL<E2.SAL				
TABLE ACCESS	EMP	FULL	5078	120
Filter Predicates				
E2.JOB='ENGINEER'				

Other XML

```
{info}
    info type="db_version"
        11.2.0.2
    info type="parse_schema"
    "SYSTEM"
    info type="plan_hash"
        1892026187
    info type="plan_hash_2"
        845382767
    {hint}
        USE_MERGE(@"SEL$1" "E2"@"SEL$1")
        LEADING(@"SEL$1" "E1"@"SEL$1" "E2"@"SEL$1")
        FULL(@"SEL$1" "E2"@"SEL$1")
        INDEX_RS_ASC(@"SEL$1" "E1"@"SEL$1" ("EMP"."JOB"))
        OUTLINE_LEAF(@"SEL$1")
        ALL_ROWS
        DB_VERSION('11.2.0.2')
        OPTIMIZER_FEATURES_ENABLE('11.2.0.2')
        IGNORE_OPTIM_EMBEDDED_HINTS
```