

Advanced Databases - LAB Week 5

Indexes and Query Optimization

Dr. Pierpaolo Dondio

In this lab we will experiment with query optimization.

Create the following tables:

1. Persons
2. Jobs
3. Jobs-persons

A list of persons is connected many-to-many to a list of jobs (the table jobs-person is the relation table).

```
drop table persons;
create table persons(
person_id integer,
person_name varchar(20),
person_surname varchar(20),
person_age integer not null,
person_wealth integer,
person_weight float
);
```

```
drop table jobs_person;
create table jobs_person(
jobs_id integer,
person_id integer,
start_date date,
end_date date);
```

```
drop table jobs;
create table jobs(
jobs_id integer,
job_description varchar(200),
salary integer
);
```

Execute the following sql 3 commands block **ONE BY ONE** to fill the tables with random data. The commands are also in the *populate.sql* (but execute them one by one!!)

```
/* 1. populate table persons */
declare v_p_id number;
v_p_name varchar2(20);
v_p_surname varchar2(20);
v_p_age integer;
p_wealth float;
p_weight float;
BEGIN
  FOR i IN 1..10000 LOOP
    select DBMS_RANDOM.STRING('a', 20) into v_p_name from dual;
    select DBMS_RANDOM.STRING('a', 20) into v_p_surname from dual;
    SELECT TRUNC(DBMS_RANDOM.VALUE(18, 100)) into v_p_age FROM DUAL;
    SELECT TRUNC(DBMS_RANDOM.VALUE(0,10000000)) into p_wealth FROM
DUAL;
    SELECT trunc(DBMS_RANDOM.VALUE(40, 120),2) into p_weight FROM
DUAL;
    insert into persons
values(i,v_p_name,v_p_surname,v_p_age,p_wealth,p_weight);
  END LOOP;
end;

/* 2. populate table jobs */
declare j_id number;
j_description varchar2(100);
j_salary float;
BEGIN
  FOR i IN 1..10000 LOOP
    select DBMS_RANDOM.STRING('a', 100) into j_description from dual;
    SELECT TRUNC(DBMS_RANDOM.VALUE(0,100000)) into j_salary FROM
DUAL;
    insert into jobs values(i,j_description,j_salary);
  END LOOP;
end;
```

```

/* 3. populate table jobs-persons */
declare j_id number;
p_id integer;
start_date date;
end_date date;
st integer;
en integer;
BEGIN
  FOR i IN 1..10000 LOOP
    FOR j in 1..15 LOOP
      SELECT TRUNC(DBMS_RANDOM.VALUE(0,1000000)) into p_id FROM DUAL;
      SELECT TRUNC(DBMS_RANDOM.VALUE(0, 1000)) into st FROM DUAL;
      SELECT TRUNC(DBMS_RANDOM.VALUE(0, 2000)) into en FROM DUAL;
      SELECT TO_DATE(TRUNC(DBMS_RANDOM.VALUE(2452641,2452641+st)), 'J')
into start_date FROM DUAL;
      SELECT
TO_DATE(TRUNC(DBMS_RANDOM.VALUE(2452641+st,2452641+st+en)), 'J') into
end_date FROM DUAL;
      insert into jobs_person values(i,p_id,start_date,end_date);
    END LOOP;
  END LOOP;
end;

```

There are no keys or indexes defined. The person_id and the jobs_id are unique.

Using the oracle function Explain Plain (in SQL developer is the fourth icon from the left in the sql script window or press F10), we will analyse how ORACLE executes queries and the cost of each query - a number expressing how much resources and time your query takes.

The screenshot shows the SQL Developer interface. The top pane displays the SQL script: `select * from persons;`. The bottom pane shows the Explain Plan for this query. The Explain Plan table has four columns: OPERATION, OBJECT_NAME, OPTIONS, and COST. It shows a single row for the SELECT STATEMENT with a cost of 248, and a sub-row for TABLE ACCESS on the PERSONS table with a cost of 248.

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			248
TABLE ACCESS	PERSONS	FULL	248

Execute the following steps to analyse Oracle Index behaviour

1. Check that data are in the three tables. Have a look at the data
There are no indexes or keys defined at this stage.
2. Execute the following query

Query1.

*select * from persons*

And select the explain plan function.

How much is the cost? _____

Was it a full or index scan of the table? _____

Why? _____

3. Execute

Query2 .

*select * from persons where person_id>1000 and person_id<3000*

Total cost? _____

Full or index scan? Why? _____

Any difference with the previous query? _____

4. Define a primary key over *person_id* (using an ALTER TABLE ... ADD CONSTRAINTS statement)
Remember that this creates an index on *person_id* as well. Perform **Query1**

Cost? _____

Full or Index? _____

Comment the results.

Perform **Query2**

Cost? _____

Full or Index? _____

Comment the results.

Query1 requires a full scan since it gives back the full unfiltered table

Query2 requires uses the index scan on the primary index *person_id* to filter the data. Note that range scan refers to accessing an interval of value (range) using an index (therefore oracle finds the starting point using the index and then the scan is sequential over an ordered list. It is therefore faster than full scan)

5. Perform the following

Query3

*select * from persons where person_id+5>1000 and person_id<3000*

Check cost and type of scan

Cost: _____

Type of Scan _____

Query4

*select * from persons where person_id+5>1000 and person_id*2<3000*

Check cost and type of scan

Cost: _____

Type of Scan _____

Comment the behaviour of Query3 and Query4

In **Query3** the index `person_id` is contained in the expression `(person_id+5)` so it cannot be used. However, the index can be used in the other where clause `(person_id<20000)`, so Oracle performs an index access of the table to get the persons with `person_id<20000` and at the same time it filters the condition `(person_id+5>20000)`

In **Query4**, none of the index can be used so a full scan is performed

Remember: if an index is used in an expression that affects the ordering of the data , it won't be used!

6. Execute

Query5.

select person_age, count(person_id) from persons group by person_age;

Cost? _____

Full or Index? _____

Comment the results.

The index on `person_id` does not help since we are grouping by `person_age`, so a full scan is required. Note the extra cost of grouping, executed by Oracle in a quick way by hashing the persons by age during the full scan

7. Define an index on `person_age` by executing:

```
create index p_age on persons(person_age);
```

Execute again **Query5**

Cost? _____

Full or Index? _____

Even if an index is defined on `person_age`, the index is not used, why?

The reason is the following (IMPORTANT!): if a column contains NULL values or it has been defined (with CREATE TABLE) without NOT NULL the index will be ineffective!

1. Drop the table `persons`.
2. Modify the create table statement adding "*not null*" to the field `person_age` (and add primary key to the `person_id` field so you do not need to alter the table afterwards).
3. Populate the table with the sql command used before (page 2, block 1)
4. Execute query5

Cost? _____

Full or Index? _____

You should see an index-like access (hash, i.e. the type of index create on `person_age` – default) and the cost is now reduced

8. Joining two tables

Perform the following query:

```
select jobs.jobs_id,jobs.job_description,jobs.salary,jobs_person.person_id
from jobs inner join jobs_person
on jobs.jobs_id = jobs_person.jobs_id
where jobs_person.jobs_id=34;
```

Cost? _____

Full or Index? _____

Comment the results.

Note how the query is divided into steps: first the full scans and the hash table used to speed-up the join.

Add indexes on jobs_id.jobs_id and jobs_person.jobs_id (note that one could be a primary key and the other a foreign key).

Check again the results.

Did they improve or not? Why?

(you should see a reduction in cost due to the usage of indexes).

9. Reduce the cost of this query as much as you can:

```
/* select person name, max salary and job description between 2003 and 2004 */  
select p.person_name, j.salary, j.job_description  
from persons p inner join jobs_person jp on p.person_id = jp.person_id  
inner join jobs j on jp.jobs_id=j.jobs_id  
where jp.start_date > '01-JAN-2003' and jp.end_date < '31-DEC-03';
```

Use indexes, temporary tables, change the SQL code, split the join – but be sure the result is still equivalent!