3rd Lab: Scrutinizing the Internals of Column-Oriented DBs

Family name: Ruiz Bonilla Given name: Cristian

Family name: Cidraque Sala Given name: Carles

Question 1

For each question in the 3rd Lab SQL quiz:

1) Explain what structures you created

Pel primer exercici la solució que hem enviat al learn es la combinació de 3 BITMAPS i una vista materialitzada per la tercera query:

- *CREATE BITMAP INDEX bitmapPobl ON poll_answers(pobl) PCTFREE 0;*
- *CREATE BITMAP INDEX bitmapEdat ON poll_answers(edat) PCTFREE 0;*
- *CREATE BITMAP INDEX bitmapCandVal ON poll_answers(cand,val) PCTFREE 0;*
- *CREATE MATERIALIZED VIEW V3 ORGANIZATION HEAP PCTFREE 0 BUILD IMMEDIATE REFRESH COMPLETE ON DEMAND ENABLE QUERY REWRITE AS (SELECT cand AS a, AVG(val) AS b FROM poll_answers GROUP BY cand);*

Amb aquesta combinació, aconseguim un cost mitjà de 11,5 utilitzant 248 blocs.

Tot i això, hem trobat que hi ha una combinació millor (que no hem enviat al learn pel factor de penalització) que es la combinació dels 3 BITMAPS anteriors, la vista materialitzada per la tercera query i una altre vista materialitzada per la primera query.

- *CREATE BITMAP INDEX bitmapPobl ON poll_answers(pobl) PCTFREE 0;*
- *CREATE BITMAP INDEX bitmapEdat ON poll_answers(edat) PCTFREE 0;*
- *CREATE BITMAP INDEX bitmapCandVal ON poll_answers(cand,val) PCTFREE 0;*
- *CREATE MATERIALIZED VIEW V3 ORGANIZATION HEAP PCTFREE 0 BUILD IMMEDIATE REFRESH COMPLETE ON DEMAND ENABLE QUERY REWRITE AS (SELECT cand AS a, AVG(val) AS b FROM poll_answers GROUP BY cand);*
- *CREATE MATERIALIZED VIEW V1 ORGANIZATION HEAP PCTFREE 0 BUILD IMMEDIATE REFRESH COMPLETE ON DEMAND ENABLE QUERY REWRITE AS (SELECT pobl AS a, MIN(edat) AS b, MAX(edat) AS c, COUNT(*) AS d FROM poll_answers GROUP BY pobl);*

Aquesta combinació aconsegueix un cost mitjà de 6,9 utilitzant 272 blocs, tot i que és possible que en funció de les dades emprades algun bitmap de pobl o d'edat ocupi 16 blocs i per tant es passa de la limitació de 275 blocs. De tota manera, en algunes execucions hem

arribat a obtenir que aquests dos bitmaps donen 8 blocs i per tant no ens passem de la limitació de 275, ja que en total ocupa 272 blocs.

***Pel segon exercici** hem fet una fragmentació vertical amb els atributs pobl, edat, cand i val.*

Primer hem creat els tipus nested_type que conté pobl, edat, cand i val.

```
CREATE OR REPLACE TYPE nested_type AS OBJECT (
        pobl INTEGER,
        edat INTEGER,
        cand INTEGER,
        val INTEGER
);
```

Després hem creat la nested table nested_type_nt:

```
CREATE OR REPLACE TYPE nested_type_nt IS TABLE OF nested_type;
```

Després ho hem juntat tot a la taula poll_answers d'aquesta manera:

```
CREATE TABLE  poll_answers(
  ref INTEGER,
  frNested nested_type_nt,
  resposta_1 VARCHAR2(10),
  resposta_2 VARCHAR2(10),
  resposta_3 VARCHAR2(10),
  resposta_4 VARCHAR2(10),
  resposta_5 VARCHAR2(10)
)NESTED TABLE frNested STORE AS nested_tab_1
PCTFREE 0 ENABLE ROW MOVEMENT;
```

Hem modificat l'script per fer les insercions respecte la nova definició de la taula poll_answers:

```
DECLARE
  j INTEGER;
  maxTuples CONSTANT INTEGER := 20000;
  maxPobl CONSTANT INTEGER := 200;
  maxEdat CONSTANT INTEGER := 100;
  maxCand CONSTANT INTEGER := 10;
  maxVal CONSTANT INTEGER := 10;
BEGIN
DBMS_RANDOM.seed(0);
-- Insertions
FOR j IN 1..(maxTuples) LOOP
    INSERT INTO poll_answers(ref,
    frNested,
    resposta_1,resposta_2,resposta_3,resposta_4,resposta_5)--,resposta_6,resposta_7,resposta_8,resposta_9,resposta_10)
    VALUES (
    j,
    nested_type_nt(nested_type(dbms_random.value(1, maxPobl), dbms_random.value(1,maxEdat),  dbms_random.value(1,
maxCand), dbms_random.value(1, maxVal))),
        LPAD(dbms_random.string('U',50),10,'-'),
        LPAD(dbms_random.string('U',50),10,'-'),
        LPAD(dbms_random.string('U',50),10,'-'),
        LPAD(dbms_random.string('U',50),10,'-'),
        LPAD(dbms_random.string('U',50),10,'-')
  );
END LOOP;
END;
```

Y finalment, hem afegit 3 bitmaps:

```
CREATE BITMAP INDEX poblbitmap ON nested_tab_1(pobl) PCTFREE 0;
CREATE BITMAP INDEX edatbitmap ON nested_tab_1(edat) PCTFREE 0;
CREATE BITMAP INDEX candvalbitmap ON nested_tab_1(cand, val) PCTFREE 0;
```

2) Explain Oracle's execution plan considering the structures you created

En aquesta part explicarem els pla d'execució de les dues solucions que hem trobat, primer comentarem la que vam enviar al Learn, és a dir, la **primera** solució en aquest document.

Pla d'execució per la primera query:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	16	120	840
HASH (GROUP BY)			16	120	840
VIEW	index$_join$_001		14	20.000	140.000
HASH JOIN			14	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	140.000
BITMAP INDEX (FULL SCAN)	BITMAPEDAT		0	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	140.000
BITMAP INDEX (FULL SCAN)	BITMAPPOBL		0	0	0

Utilitza els bitmaps d'edat i de pobl ya que només accedim a aquests dos atributs en aquesta query. Després fa la hash join per unir les dades.

Pla d'execució per la segona consulta:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	30	20.000	260.000
HASH (GROUP BY)			30	20.000	260.000
VIEW	index$_join$_001		29	20.000	260.000
HASH JOIN			21	0	0
HASH JOIN			14	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPEDAT		0	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPINDEX3		0	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPPOBL		0	0	0

En aquest cas utilitza els 3 bitmaps creats ja que accedeix a edat, pobl, cand i val, per tant per agafar les dades sense anar a la taula ha d'utilitzar tots els índex. Primer veiem que fa un hash join amb les dades d'edat i del bitmapindex3 que conté cand i val. Després fa un hash join amb l'últim bitmap de pobl.

Pla d'execució per la tercera consulta:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	5	10	250
MAT_VIEW REWRITE ACCESS (FULL)	V3	ANALYZED	5	10	250

A la tercera consulta sí que pot utilitzar una vista materialitzada i es el que fa.

Plans d'execució de la **segona** solució **pel primer exercici:**

Pla d'execució per la primera consulta:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	5	120	1.680
MAT_VIEW REWRITE ACCESS (FULL)	V1	ANALYZED	5	120	1.680

En aquest cas utilitza directament la vista materialitzada per la primera query i per tant dóna un resultat molt més inferior que a la solució anterior. Passem d'un cost de 16 a 5.

Pla d'execució per la segona consulta:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	24	20.000	260.000
HASH (GROUP BY)			24	20.000	260.000
VIEW	index$_join$_001		23	20.000	260.000
HASH JOIN			17	0	0
HASH JOIN			11	0	0
BITMAP CONVERSION (TO ROWIDS)			5	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPEDAT		0	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPINDEX3		0	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	260.000
BITMAP INDEX (FULL SCAN)	BITMAPPOBL		0	0	0

El pla d'execució es el mateix que a la solució anterior, la única diferencia es que acabem amb un cost inferior, passem de 30 a 24. L'explicació d'això és que les dades son random, per tant pot variar el cost en funció de les dades introduïdes.

Pla d'execució per la tercera consulta:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	5	10	250
MAT_VIEW REWRITE ACCESS (FULL)	V3	ANALYZED	5	10	250

De nou, aquí utilitza la vista materialitzada de la tercera query i dona un cost molt baix.

Per tant, sense tenir en compte el segon pla d'execució que es variable en funció de les dades, **podem concloure** que aquesta segona solució és millor que la primera sobretot per la primera query ja que utilitza una vista materialitzada per la primera query.

Pla d'execució per el segon exercici:

Pla d'execució de la primera query:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	93	200	4.800
HASH (GROUP BY)			93	200	4.800
VIEW	index$_join$_003		92	20.000	480.000
HASH JOIN			103	0	0
HASH JOIN			13	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	480.000
BITMAP INDEX (FULL SCAN)	EDATBITMAP		0	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	480.000
BITMAP INDEX (FULL SCAN)	POBLBITMAP		0	0	0
INDEX (FAST FULL SCAN)	SYS_FK0001381763N00002$	ANALYZED	90	20.000	480.000

En aquest cas veiem que utilitza els bitmaps que hem creat per l'atribut pobl i per l'atribut edat ja que només accedeix a aquests atributs.

Pla d'execució per la segona query:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	269	20.000	600.000
HASH (GROUP BY)			269	20.000	600.000
VIEW	index$_join$_003		104	20.000	600.000
HASH JOIN			109	0	0
HASH JOIN			19	0	0
HASH JOIN			12	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	600.000
BITMAP INDEX (FULL SCAN)	CANDVALBITM...		0	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	600.000
BITMAP INDEX (FULL SCAN)	EDATBITMAP		0	0	0
BITMAP CONVERSION (TO ROWIDS)			7	20.000	600.000
BITMAP INDEX (FULL SCAN)	POBLBITMAP		0	0	0
INDEX (FAST FULL SCAN)	SYS_FK0001381...	ANALYZED	90	20.000	600.000

A la segona query s'accedeix a pobl, edat, cand i val i per tant utilitza els 3 bitmaps que hem definit. Tot i això el cost és molt dolent degut a la quantitat de hash joins que ha de fer Oracle.

Pla d'execució per la tercera query:

Operación	Objeto	Optimizador	Coste	Cardinalidad	Bytes
SELECT STATEMENT		ALL_ROWS	80	10	230
HASH (GROUP BY)			80	10	230
VIEW	index$_join$_003		78	20.000	460.000
HASH JOIN			96	0	0
BITMAP CONVERSION (TO ROWIDS)			6	20.000	460.000
BITMAP INDEX (FULL SCAN)	CANDVALBITM...		0	0	0
INDEX (FAST FULL SCAN)	SYS_FK0001381...	ANALYZED	90	20.000	460.000

Per últim, a la tercera query, només s'utilitza el bitmap compost per cand i val ja que només accedim a aquests atributs.

Com veiem, el problema principalment ve donat dels hash joins que fa Oracle que fan empitjorar molt el cost.

3) For the execution plan given, discuss what steps would be executed differently by a column-oriented databases (refer to the physical data structures as well as query processing techniques used)

Primer s'hauria de generar un fragment per cada atribut. Sobre aquests fragments acabariem tenint 3 vectors, el diccionari, el vector de final d'índexs i el vector de posicions. Com que l'ordre de les instàncies es manté a les tres estructures, no necessita guardar PKs ni fer cap join. Senzillament accedeix a les taules de hash i fa l'operació AND.

Pel segon exercici, una column-oriented funcionaria molt diferent ja que en aquest cas si que es beneficiaria de l'affinity matrix i la fragmentació ja seria diferent, tindriem dues nested table una per {pobl,edat} i una altre per {cand,val} i per tant seria molt menys costós accedir a les dades ja que per exemple a la primera query no necessitem la informació de cand i val, mentres que a la tercera query tampoc necessitem la informació de pobl i edat. Pel cas de la segona query, també ens aniria bé.

Per últim, comentar que accedint als 3 vectors, encara seria molt més ràpid que a la row-oriented ja que no hem de descomprimir la informació.

Question 2

For the second exercise, use the affinity matrix method to decide how to fragment the database vertically. Consider now your solution for Exercise 2 in the 3rd Lab SQL quiz. Is Oracle yielding the best result when using the same vertical fragmentation strategy as suggested by the affinity matrix method? **Justify your answer.**

Per tal d'obtenir l'affinity matrix i decidir com fragmentar la base de dades verticalment, primer de tot elaborem la matriu d'utilització per a la taula que utilitzem en les queries, únicament la taula *poll_answers*.

(20%) SELECT pobl, MIN(edat), MAX(edat), COUNT(*) FROM poll_answers GROUP BY pobl;

(30%) SELECT pobl, edat, cand, MAX(val), MIN(val), AVG(val) FROM poll_answers GROUP BY pobl, edat, cand;

(50%) SELECT cand, AVG(val) FROM poll_answers GROUP BY cand;

	pobl	edat	cand	val	resposta_1	resposta_2	resposta_3	resposta_4	resposta_5
Q1(20%)	1	1	0	0	0	0	0	0	0
Q2(30%)	1	1	1	1	0	0	0	0	0
Q3(50%)	0	0	1	1	0	0	0	0	0

Un cop la tenim ja podem elaborar l'affinity matrix:

	pobl	edat	cand	val	resp_1	resp_2	resp_3	resp_4	resp_5
pobl	50	50	30	30	-	-	-	-	-
edat	50	50	30	30	-	-	-	-	-
cand	30	30	80	80	-	-	-	-	-
val	30	30	80	80	-	-	-	-	-
resp_1	-	-	-	-	-	-	-	-	-
resp_2	-	-	-	-	-	-	-	-	-
resp_3	-	-	-	-	-	-	-	-	-
resp_4	-	-	-	-	-	-	-	-	-
resp_5	-	-	-	-	-	-	-	-	-

A partir de l'affinity matrix veiem que les relacions més destacades son (pobl, edat) amb un 50%, veiem que pobl i edat el 100% de les vegades apareixen junts.
L'altre relació més destacada es (cand, val) amb un 80%, veiem que cand i val el 100% de les vegades apareixen junts.
Per tant hem de realitzar dos fragments: P1: pobl, edat i P2: cand, val

Per cadascun dels fragments decidits a fer cal fer una nested table, per tant, és una fragmentació híbrida, ja que els dos fragments contenen més d'un atribut, més d'una columna de la taula original.

Veiem que Oracle com està clar quan utilitzem l'objecte nested table no fa fragmentació vertical de forma nativa, sino que la simula amb aquests objectes.
Teòricament això es igual a una fragmentació vertical, però si mirem a la pràctica a nivell físic no es el mateix, això ens fa obtindre resultats molt diferents als esperats.

Veiem que un cop s'han resolt els índexs del resultat s'hi fa una HASH JOIN de la nested table, ja que per reconstruir les tuples amb fragmentació vertical es fa amb *join*. Oracle manté un índex per ser capaç de desfer la fragmentació vertical (es part de la fragmentació vertical), es el índex on estàn les PK que fan join amb la taula original. En una column-oriented no és necessari aquest join tant costós, ens l'estalviem, ja que amb l'ordre en que estàn les tuples ja és suficient.

Per tant, optem per fer una única nested table ja que el gran cost ve donat per les joins que acabem de comentar. La millor solució és precisament evitar aquestes joins, fent una nested table amb tots els atributs que facin servir les diferents queries.

Veiem per tant que una database row-oriented com Oracle a més de no fer una fragmentació "real" no es beneficia tampoc de simular-la.