DES3UE Datenbanksysteme WS 2018 Übung 11

Abgabetermin: 19.12.2018, 13:30 Uhr

☐ DES31UE Niklas **Name** Niklas Vest **Aufwand in h** 6
☐ DES32UE Niklas
☒ DES33UE Traxler **Punkte** _____ **Kurzzeichen Tutor** _____

Ziel dieser Übung ist den Einsatz von Indizes kennenzulernen und die Optimierung von Abfragen auszuprobieren. Thema sind auch die Verwendung von Optimizer Hints und theoretische Aspekte der Optimierung.

1. Indizes (5 Punkte)

Mit der Verwendung von Indizes kann die Datenbank den Zugriff auf einzelne Datensätze optimieren. Ob ein Index verwendet wird, wird vom Optimizer unter der Berücksichtigung mehrerer Kriterien bestimmt.

1. Erstellen Sie eine Tabelle customer_detail, die für alle Kunden die ID, Vor- und Nachname, E-Mail-Adresse, Bezirk (district), Postleitzahl, Telefonnummer, Stadt und Land enthält. Legen Sie für diese Tabelle vorerst keinen Primärschlüssel bzw. Index fest.
2. Erstellen Sie ein SQL-Statement, das alle Datensätze aus der Tabelle customer_detail auflistet, deren Nachnamen mit ‚MAR' beginnt (verwenden Sie ein LIKE). Analysieren Sie den Ausführungsplan.
3. Setzen Sie auf die Spalte last_name einen Index und führen Sie das vorherige Statement erneut aus, untersuchen Sie den Ausführungsplan auf Unterschiede.
4. Ändern Sie das unter Punkt 1.2 erstellte Statement, sodass der Vergleich nicht mehr mit einem LIKE durchgeführt wird, sondern mittels SUBSTR auf den last_name. Analysieren Sie den Ausführungsplan auf die Verwendung des bereits erstellten Index.
5. Erstellen Sie einen Function-based Index, der das SUBSTR auf last_name berücksichtigt. Wiederholen Sie die Ausführung des vorigen Statements, vergleichen Sie den Ausführungsplan.

2. Optimizer Hints (2 Punkte)

Sie können die Verwendung eines Index auch durch einen Optimizer Hint erzwingen.

1. Legen Sie zusätzlich zu den beiden Indizes von Beispiel 1 einen Index auf die Spalte country. Fragen Sie mit einem SQL-Statement alle Personen aus ‚India' ab, deren Nachname mit ‚MAR' beginnt. Analysieren Sie den Ausführungsplan ob bzw. welcher Index verwendet wird.
2. Verwenden Sie einen Optimizer Hint und erzwingen Sie damit die Verwendung des Index auf country. Kontrollieren Sie im Ausführungsplan, ob der gewünschte Index verwendet wird.

3. Optimierung von SQL-Statements (11 Punkte – 3 + 4 + 4 Punkte)

Beachten Sie bei der Optimierung von SQL-Statements, dass die Ergebnismenge des weniger performanten Statements der Ergebnismenge des optimierten Statements entsprechen muss. Verwenden Sie zur Überprüfung den MINUS-Operator. Sie benötigen für diese Aufgabe keine Optimizer Hints.

1. Selektieren Sie alle Kunden, die einen Film der Kategorien ‚Comedy', ‚Family' oder ‚Children' ausgeliehen haben und deren Film kürzer als 100 Minuten ist, fügen Sie der Menge zusätzlich alle Kunden hinzu, die einen Film der Kategorien ‚Classics' oder ‚Animation' geliehen haben; vermeiden Sie Duplikate. Optimieren Sie das gegebene Statement und vergleichen Sie die Ausführungsplane. Tipp: Erstellen Sie eine geeignete WHERE Bedingung, vermeiden Sie DISTINCT.

```
SELECT DISTINCT customer_id, first_name, last_name
FROM customer
  INNER JOIN rental USING (customer_id)
  INNER JOIN inventory USING (inventory_id)
  INNER JOIN film USING (film_id)
  INNER JOIN film_category USING (film_id)
  INNER JOIN category USING (category_id)
WHERE name IN ('Comedy', 'Family', 'Children') AND length < 100
UNION
SELECT DISTINCT customer_id, first_name, last_name
FROM customer
  INNER JOIN rental USING (customer_id)
  INNER JOIN inventory USING (inventory_id)
  INNER JOIN film USING (film_id)
  INNER JOIN film_category USING (film_id)
  INNER JOIN category USING (category_id)
WHERE name IN ('Classics', 'Animation');
```

2. Das gegebene SQL-Statement verwendet korrelierte Sub-Selects um den Umsatz der einzelnen Jahre pro Kunde zu berechnen, für jeden Kunden wird auch der Gesamtumsatz ausgegeben. Optimieren Sie das Statement, vergleichen und interpretieren Sie die Ausführungspläne. Tipp: Verwenden Sie GROUPING SETS und PIVOT.

```
SELECT c.customer_id, first_name, last_name,
       (SELECT SUM(amount)
        FROM payment
        WHERE c.customer_id = customer_id
          AND to_char(payment_date, 'yy') = '13') AS umsatz13,
       (SELECT SUM(amount)
        FROM payment
        WHERE c.customer_id = customer_id
          AND to_char(payment_date, 'yy') = '14') AS umsatz14,
       (SELECT SUM(amount)
        FROM payment
        WHERE c.customer_id = customer_id
          AND to_char(payment_date, 'yy') = '15') AS umsatz15,
       (SELECT SUM(amount)
        FROM payment
        WHERE c.customer_id = customer_id) AS umsatzGesamt
FROM customer c;
```

3. Die Funktion num_longer_films_in_cat berechnet für eine gegebene film_id die Anzahl jener Filme in der gleichen Kategorie, die eine längere Dauer besitzen. Beachten Sie bei der Optimierung des Konstrukts, dass die Laufzeit/Komplexität der Funktion nicht im Ausführungsplan aufscheint, messen Sie für dieses Beispiel die Laufzeit des SQL-Statements, das die Funktion verwendet. Erstellen Sie ein besseres SQL-Statement und vergleichen Sie die Laufzeit. Tipp: Verwenden Sie eine analytische Funktion in Kombination mit RANGE statt der PL/SQL-Funktion.

```
CREATE OR REPLACE FUNCTION num_longer_films_in_cat (filmid IN NUMBER)
RETURN NUMBER
AS
  categoryid NUMBER;
  len NUMBER;
  numfilms NUMBER := 0;
BEGIN
  SELECT category_id, length INTO categoryid, len
  FROM film INNER JOIN film_category USING (film_id)
  WHERE film_id = filmid;

  FOR film IN (SELECT *
               FROM film INNER JOIN film_category USING (film_id)
               WHERE category_id = categoryid AND length > len) LOOP
    numfilms := numfilms + 1;
  END LOOP;

  RETURN numfilms;
END;
/

SELECT film_id, title, num_longer_films_in_cat(film_id) AS longerFilmsInCategory
FROM film;
```

4. Theorie und Interpretation **(6 Punkte** – je 2 Punkte**)**

1. Welche Aussagen zu Tuning treffen zu?
   - [x] Tuning von einzelnen SQL-Statements (= Umschreiben) bringt die meiste Performance-Verbesserung.
   - [x] Die Konfiguration des DBMS und die Leistungsfähigkeit des Betriebssystems sind wichtige Voraussetzungen für performante Abfragen.
   - [ ] Ein Großteil der Optimierung von SQL-Statements passiert automatisch durch den Query-Optimizer.
   - [ ] Tuning beginnt man am besten mit dem Setzen eines Index.

2. Welche Aussagen zu Indizes treffen zu?
   - [ ] Wenn ein Index auf ein Attribut gesetzt ist und dieses Attribut wird im SQL-Statement abgefragt, kann man davon ausgehen, dass der Index auch benutzt wird.
   - [x] Ein Index erzeugt Overhead beim Einfügen und Ändern von Datensätzen.
   - [x] Eine Tabelle, die vollständig im Index liegt (index-organized table) bietet die besten Zugriffswerte, benötigt allerdings zusätzlichen Speicherplatz.
   - [x] Ein Attribut kann in mehreren Indizes vorkommen (in Kombination mit anderen Attributen).

3. Welche Aussagen zum Applikations-Design treffen zu?
   - [x] Joins und Berechnungen innerhalb der Daten sollen möglichst am Server und nicht in der Client-Anwendung passieren.
   - [x] Sperren von langen Transaktionen (zB Benutzerinteraktionen) können andere Transaktionen beeinträchtigen und die Antwortzeiten (Transaktion wartet) massiv erhöhen.
   - [ ] Komplexe Abfragen, die rein lesend auf die Daten zugreifen (analytisch), können ohne Einschränkungen auch während den Hauptbetriebszeiten auf die Datenbank abgesetzt werden, da sie keine Sperren anfordern.
   - [ ] Für jede SQL-Abfrage soll die Applikation (Client) eine neue Verbindung zur Datenbank aufbauen, besser mehrere kleine SELECTs mit wenigen Daten als ein großes SELECT mit vielen Daten.

Ausarbeitung UE11

1. Indizes

```sql
-- 1.1
CREATE TABLE customer_detail AS
SELECT customer_id,
       first_name,
       last_name,
       email,
       phone,
       district,
       postal_code,
       city,
       country
FROM customer
       INNER JOIN address USING (address_id)
       INNER JOIN city USING (city_id)
       INNER JOIN country USING (country_id);

-- 1.2
SELECT *
FROM customer_detail
WHERE last_name LIKE 'MAR%';
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                |     1 |    98 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUSTOMER_DETAIL|     1 |    98 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("LAST_NAME" LIKE 'MAR%')
```

```sql
-- 1.3
CREATE INDEX customer_detail_lname ON customer_detail (last_name);

SELECT *
FROM customer_detail
WHERE last_name LIKE 'MAR%';
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                     |     1 |    98 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER_DETAIL     |     1 |    98 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | CUSTOMER_DETAIL_LNAME |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("LAST_NAME" LIKE 'MAR%')
       filter("LAST_NAME" LIKE 'MAR%')
```

Illustration 1: Ausführungsplan berücksichtigt erstellten Index!

```sql
--1.4
SELECT *
FROM customer_detail
WHERE SUBSTR(last_name, 0, 3) = 'MAR';
```

```
-------------------------------------------------------------------------------------
| Id  | Operation         | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                 |     6 |   588 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUSTOMER_DETAIL |     6 |   588 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(SUBSTR("LAST_NAME",0,3)='MAR')
```

Illustration 2: SUBSTR benötigt trotz Index einen full table access!

```sql
--1.5
CREATE INDEX customer_detail_lname_substr ON customer_detail (SUBSTR(last_name, 0, 3));

SELECT *
FROM customer_detail
WHERE SUBSTR(last_name, 0, 3) = 'MAR';
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                             |     6 |   588 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER_DETAIL             |     6 |   588 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | CUSTOMER_DETAIL_LNAME_SUBSTR|     2 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SUBSTR("LAST_NAME",0,3)='MAR')
```

Illustration 3: Neuer function-based Index wird zur Optimierung herangezogen.

2. Optimizer Hints

```sql
-- 2.1
CREATE INDEX customer_detail_country ON customer_detail (country);

SELECT *
FROM customer_detail
WHERE country = 'India'
  AND last_name LIKE 'MAR%';
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                       |     1 |    98 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER_DETAIL       |     1 |    98 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | CUSTOMER_DETAIL_LNAME |     1 |       |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("COUNTRY"='India')
   2 - access("LAST_NAME" LIKE 'MAR%')
       filter("LAST_NAME" LIKE 'MAR%')
```

Illustration 4: Lt. Ausführungsplan wird der neu erstellte Index nicht verwendet.

```sql
-- 2.2
SELECT /*+ INDEX (customer_detail customer_detail_country)*/ *
FROM customer_detail
WHERE country = 'India'
  AND last_name LIKE 'MAR%';
```

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                       |     1 |    98 |     4   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER_DETAIL       |     1 |    98 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | CUSTOMER_DETAIL_COUNTRY |     6 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("LAST_NAME" LIKE 'MAR%')
   2 - access("COUNTRY"='India')
```

Illustration 5: Mit explizitem Hinweis verwendet der Query optimizer den neuen Index!

3. Optimierung von SQL-Statements

Anmerkung: Gleichheit der Ergebnistupelmengen der jeweils unoptimierten und optimierten Versionen werden nur in der anliegenden SQL Datei getestet und in diesem Dokument nicht angeführt.

```sql
-- 3.1
SELECT customer_id, first_name, last_name
FROM customer
       INNER JOIN rental USING (customer_id)
       INNER JOIN inventory USING (inventory_id)
       INNER JOIN film USING (film_id)
       INNER JOIN film_category USING (film_id)
       INNER JOIN category USING (category_id)
WHERE (name IN ('Comedy', 'Family', 'Children') AND length < 100)
   OR name IN ('Classics', 'Animation');
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3195	174K	108 (4)	00:00:01
1	SORT UNIQUE		3195	174K	108 (4)	00:00:01

Illustration 6: Abfrage 3.1 ohne Optimierung

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3002	178K	57 (2)	00:00:01
* 1	HASH JOIN		3002	178K	57 (2)	00:00:01

Illustration 7: Abfrage 3.1 mit Optimierung

```sql
-- 3.2
SELECT *
FROM (
       SELECT COALESCE(to_char(payment_date, 'yy'), 'total') AS year,
              customer_id,
              first_name,
              last_name,
              SUM(amount)                                    AS umsatz
       FROM customer
              INNER JOIN payment USING (customer_id)
       GROUP BY GROUPING SETS (
         (to_char(payment_date, 'yy'), customer_id, first_name, last_name),
         (customer_id, first_name, last_name)
         )
     )
       PIVOT (
              AVG(umsatz)
              FOR year
              IN ('13' AS umsatz13,
                '14' AS umsatz14,
                '15' AS umsatz15,
                'total' AS umsatzgesamt)
    );
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	123	111 (1)	00:00:01
* 1	HASH JOIN OUTER		1	123	111 (1)	00:00:01

Illustration 8: Abfrage 3.2 ohne Optimierung

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		16049	564K	33 (4)	00:00:01
1	SORT GROUP BY NOSORT PIVOT		16049	564K	33 (4)	00:00:01

Illustration 9: Abfrage 3.2 mit Optimierung

```sql
-- 3.3
SELECT film_id,
       title,
       RANK() OVER (
         PARTITION BY category_id
         ORDER BY length DESC) - 1 AS longerfilmsincategory
FROM film
       INNER JOIN film_category USING (film_id)
ORDER BY film_id;
```

```
500 rows retrieved starting from 1 in 555 ms (execution: 112 ms, fetching: 443 ms)
```

Illustration 10: Abfrage 3.3 ohne Optimierung

```
500 rows retrieved starting from 1 in 162 ms (execution: 28 ms, fetching: 134 ms)
```

Illustration 11: Abfrage 3.3 mit Optimierung