Gruppe A	PRÜFUNG AUS DATENBANKSYSTEME **VL 181.186**		14. 1. 2010
Kennnr.	Matrikelnr.	Familienname	Vorname

Arbeitszeit: 120 Minuten. Aufgaben sind auf den Angabeblättern zu lösen; Zusatzblätter werden nicht gewertet.

Aufgabe 1: Kostenabschätzungen (20)

Eine Universitätsdatenbank enthält folgende Relationen:

Student(SNr, Name, Alter, Wohnort, Email) (kurz *s*),
Vorlesung(VNr, Name, Professor, Kategorie) (kurz *v*) und
Prüfung(SNr, VNr, Versuchsanzahl, Note) (kurz *p*).

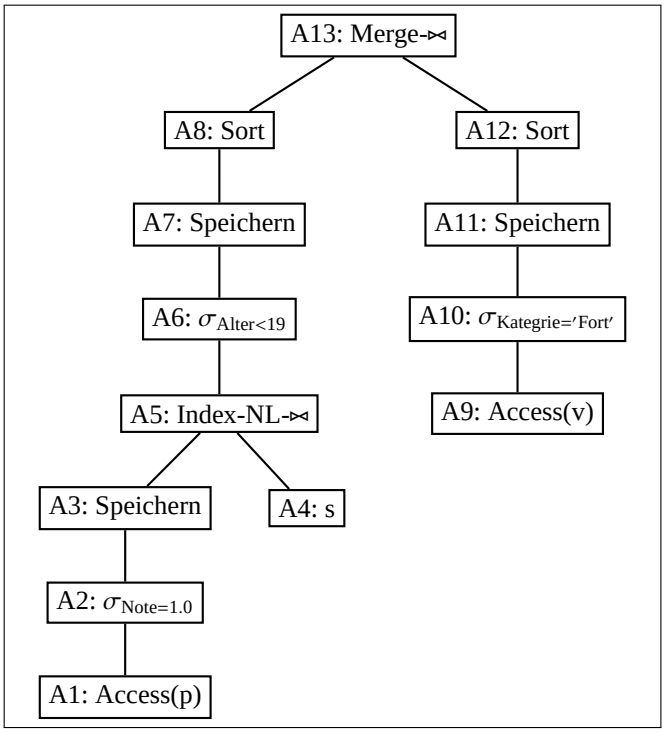
Nehmen Sie an, dass $|s| = 40000$, $|v| = 1000$, und $|p| = 600000$. Für die durchschnittlichen Tupelgrößen von *s*, *v* und *p* sind die Werte 100, 200 und 50 Bytes anzunehmen. Nehmen Sie weiters an, dass pro Seite 2000 Bytes an Nutzinformation gespeichert werden können und dass die Hauptspeicher-Puffergröße 128 Seiten beträgt. Es ist die Anfrage

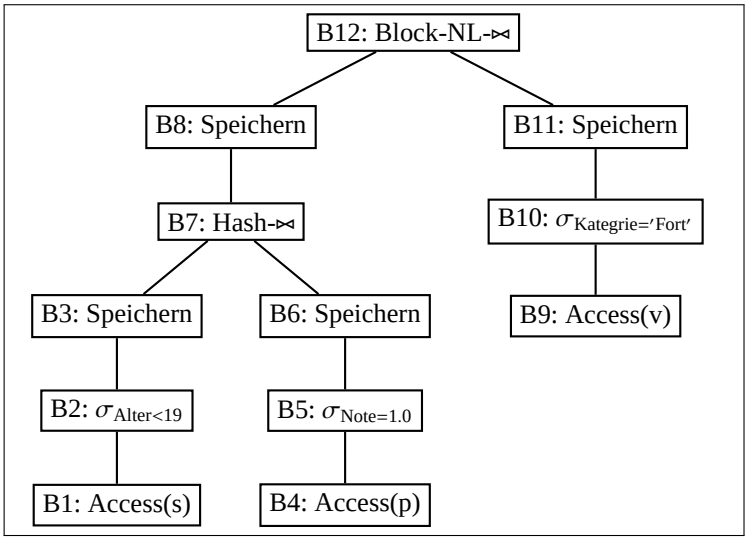
```
select *
from Student s, Vorlesung v, Prüfung p
where s.SNr = p.SNr
and v.VNr = p.VNr
and v.Kategorie = 'Fort'
and s.Alter < 19
and p.Note = 1.0;
```

auszuführen (d.h. gesucht sind Informationen über junge StudentInnen, die in einer fortgeschrittenen Vorlesung die Note "Sehr Gut (1.0)" bekommen haben).

Es sind folgende Selektivitäten anzunehmen: $Sel_{s/p} = 1/40000 = 0.000025$, $Sel_{v/p} = 1/1000 = 0.001$, $Sel_{v.Kategorie='Fort'} = 0.4$, $Sel_{s.Alter<19} = 0.2$, und $Sel_{p.Note=1.0} = 0.1$. Für die Primärschlüssel der Relationen s und v sei jeweils ein Hash-Index vorhanden. Nehmen Sie an, dass das Auslesen eines einzelnen Tupels mit einem Hash-Index durchschnittliche Kosten von 1.4 Page I/O erfordert.

Für diese Anfrage sind die Operator-Bäume für 2 Auswertungspläne gegeben: Plan A im linken Kästchen und Plan B im rechten Kästchen. Nehmen Sie bei den Block Nested Loop Joins an, dass $k = 1$ gilt und dass die äußere Relation jeweils links im Baum steht – unabhängig davon, ob dies optimal ist oder nicht. Bei der Berechnung der benötigten Seiten zum Speichern einer Relation dürfen Sie vereinfachend annehmen, dass die Tupel nicht unbedingt vollständig auf einer Seite Platz haben müssen.





(a) Berechnen Sie für jeden Knoten im Operatorbaum des Auswertungsplans A eine Abschätzung für die Anzahl der Tupel im Resultat, die Tupelgröße, die Anzahl der Seiten im Resultat, und die Kosten (Page I/O). Für das Sortieren und für Joinoperationen ist auch noch die passende Kostenformel anzugeben. Tragen Sie Ihre Berechnungen in folgende Tabelle ein.

Knoten# i	Anzahl Tupel T_i	Tupel-größe g_i	Anzahl Seiten b_i	Kostenformel	Kosten (Page I/O)
A1	600000	50	15000	-	15000 .....
A2	60000	50	1500	-	0 .....
A3	60000	50	1500	-	1500 .....
A4	40000	100	20000	-	0 .....
A5	60000	150	4500	$b_3 + 1.4 * T_3$	85500 .....
A6	12000	150	900	-	0 .....
A7	12000	150	900	-	900 .....
A8	12000	150	900	$2 * b_7 * (1 + I)$ mit ..... $I = \lceil \log_{127}(\lceil b_7/128 \rceil) \rceil = 1$	3600 .....
A9	1000	200	100	-	100 .....
A10	400	200	40	-	0 .....
A11	400	200	40	-	40 .....
A12	400	200	40	$2 * b_{11} * (1 + I)$ mit ..... $I = \lceil \log_{127}(\lceil b_{11}/128 \rceil) \rceil = 0$	80 .....
A13	4800	350	840	$(b_8 + b_{12})$	940 .....

Kosten insgesamt (Page I/O):

15000 + 1500 + 85500 + 900 + 3600 + 100 + 40 + 80 + 940 = 107660 .....

(b) Berechnen Sie für jeden Knoten im Operatorbaum des Auswertungsplans B eine Abschätzung für die Anzahl der Tupel im Resultat, die Tupelgröße, die Anzahl der Seiten im Resultat, und die Kosten (Page I/O). Für die Joinoperationen ist auch noch die passende Kostenformel anzugeben. Tragen Sie Ihre Berechnungen in folgende Tabelle ein.

Knoten# i	Anzahl Tupel T_i	Tupel-größe g_i	Anzahl Seiten b_i	Kostenformel	Kosten (Page I/O)
B1	40000	100	2000	-	2000 .....
B2	8000	100	400	-	0 .....
B3	8000	100	400	-	400 .....
B4	600000	50	15000	-	15000 .....
B5	60000	50	1500	-	0 .....
B6	60000	50	1500	-	1500 .....
B7	12000	150	900	$3 * (b_3 + b_6)$	5700 .....
B8	12000	150	900	-	900 .....
B9	1000	200	100	-	100 .....
B10	400	200	40	-	0 .....
B11	400	200	40	-	40 .....
B12	4800	350	840	$b_8 + 1 + \lceil b_8/126 \rceil * (b_{11} - 1)$	1213 .....

Kosten insgesamt (Page I/O):

2000 + 400 + 15000 + 1500 + 5700 + 900 + 100 + 40 + 1213 = 26853

Aufgabe 2: Mehrbenutzersynchronisation (10)

In einem DBMS ist eine Datenbank mit den Tabellen A und B implementiert, die als Spalten jeweils eine numerische ID ('id', Primary Key) und einen numerischen Wert ('wert') haben.

Gehen Sie davon aus, dass das DBMS ein striktes Zwei-Phasen-Sperrprotokoll verwendet wird und alle Isolation Levels gesondert implementiert sind (d.h. dirty/unrepeatable/phantom reads sind bei den jeweiligen Isolation Levels möglich).

Gegeben sind zwei Transaktionen:

Transaktion 1:

```
SELECT * FROM A;
SELECT * FROM B;
COMMIT;
```

Transaktion 2:

```
UPDATE B SET wert = 200;
UPDATE A SET wert = 100;
COMMIT;
```

(a) Bei welchen Isolation Levels kann es zu einem Deadlock kommen?

◯ Read Uncommited ◯ Read Commited ⊗ Repeatable Read ⊗ Serializable

(b) Wie müssten Sie Transaktion 2 verändern, damit es bei keinem der vier Isolation Levels zu einem Deadlock kommen kann, das Resultat aber das selbe bleibt?

```
UPDATE A SET wert = 100;
UPDATE B SET wert = 200;
COMMIT;
```

Zusätzlich zu den Transaktionen 1 und 2 (in der unveränderten Version) wird nun auch folgende Transaktion 3 ausgeführt:

```
UPDATE A SET wert = 300 WHERE id = 1;
UPDATE B SET wert = 200 WHERE id = 1;
COMMIT;
```

(c) Bei welchen Isolation Levels kann es nun zu einem Deadlock kommen?
☒ Read Uncommited ☒ Read Commited ☒ Repeatable Read ☒ Serializable

(d) Bei welchen Isolation Levels kann es nicht mehr zu kaskadierenden Rollback kommen?
○ Read Uncommited ☒ Read Commited ☒ Repeatable Read ☒ Serializable

Aufgabe 3: Sequence (6)

Definieren Sie eine *Sequence* mit Namen “primKey”, die bei 1 startet und jeweils um 1 erhöht wird. [3]

```
CREATE SEQUENCE primKey
  START WITH 1 INCREMENT BY 1;
```

Fügen Sie zwei Zeilen in die Tabelle `Row`(id, name) ein. Der Wert von `id` soll von der Sequence “primKey” kommen. [2]

```
INSERT INTO Row VALUES
  (nextval('primKey'), 'Erste Zeile');
INSERT INTO Row VALUES
  (nextval('primKey'), 'Zweite Zeile');
```

Löschen Sie die Sequence “primKey”. [1]

```
DROP SEQUENCE primKey;
```

Aufgabe 4: Hierarchische Abfragen (6)

Dieser Aufgabe liegt folgendes Schema zugrunde:

`produkt` (pid, name)
`bauplan` (produkt: produkt.pid, teil: produkt.pid, anzahl)

Evaluieren Sie folgendes SQL-Statement auf die angegebenen Tabellen, und geben Sie die Ausgabe an:

pid	name
1	’Maschine’
2	’Zahnrad’
3	’Motor’
4	’Fahrzeug’

produkt	teil	anzahl
1	2	3
1	3	1
3	2	4
4	1	2
4	3	1

```
WITH RECURSIVE temp(produkt, teil, anzahl) AS (
  SELECT produkt, teil, anzahl
  FROM bauplan
  WHERE produkt = 1
  UNION ALL
  SELECT bp.produkt, bp.teil, bp.anzahl
  FROM temp at, bauplan bp
  WHERE at.teil = bp.produkt
)
SELECT teil, name, SUM(anzahl)
FROM temp, produkt
where produkt.pid = teil
GROUP BY teil, name;
```

teil	name	sum
2	Zahnrad	7
3	Motor	1

Aufgabe 5: PL/SQL Trigger (9)

Nehmen Sie an, dass eine Datenbank wie folgt definiert wurde.

```
CREATE TABLE person (
  pid INTEGER PRIMARY KEY,
  name VARCHAR(255),
  version INTEGER
);

CREATE FUNCTION fTrigger1()
RETURNS trigger AS $$
BEGIN
  CASE NEW.pid
    WHEN 1,2 THEN
      RETURN NULL;
    WHEN 3 THEN
      NEW.name = 'Peter Pan';
    ELSE
      NULL;
  END CASE;
  NEW.version = 0;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION fTrigger2()
RETURNS trigger AS $$
BEGIN
  IF (OLD.name != NEW.name) THEN
    NEW.version = OLD.version + 1;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tTrigger1
BEFORE INSERT ON person
FOR EACH ROW EXECUTE PROCEDURE fTrigger1();

CREATE TRIGGER tTrigger2
BEFORE UPDATE ON person
FOR EACH ROW EXECUTE PROCEDURE fTrigger2();
```

Geben Sie an, wie die Tabelle person nach jedem der folgenden Befehlsblöcke aussieht.

```
INSERT INTO person VALUES (1, 'Eins');
INSERT INTO person VALUES (2, 'Zwei');
INSERT INTO person VALUES (3, 'Drei');
INSERT INTO person VALUES (4, 'Vier');
```

pid	name	version
3	Peter Pan	0
4	Vier	0

```
UPDATE person SET name = 'Vier' WHERE pid = 4;
```

pid	name	version
3	Peter Pan	0
4	Vier	0

```
UPDATE person SET name = 'Fünf' WHERE pid = 4;
```

pid	name	version
3	Peter Pan	0
4	Fünf	1

Aufgabe 6: (15)

Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind.

1. Jede Bedingung eines FOREIGN KEY Constraints kann auch mittels eines CHECK Constraints überprüft werden.

   wahr ○ falsch ⊗

2. Alle Constraints, die per ALTER TABLE Statement angelegt werden, könnten auch sofort im CREATE TABLE Statement angelegt werden.

   wahr ○ falsch ⊗

3. In PL/pgSQL-Blöcken haben in SQL-Statements Variablennamen immer Präferenz über andere Bezeichner wie Spalten- oder Tabellennamen.

   wahr ⊗ falsch ○

4. Bei FOR EACH STATEMENT Triggern enthalten die Variablen OLD und NEW ein Array aller geänderten Datensätze.

   wahr ○ falsch ⊗

5. Verwendet man das JDBC DataSource Interface um Datenbankverbindungen aufzubauen, so wird dadurch auch immer Connection Pooling angewandt.

   wahr ○ falsch ⊗

6. Prepared Statements dienen bei JDBC unter Anderem dazu, die Performance der Anfragen zu steigern.

   wahr ⊗ falsch ○

7. Wird ein RAID-System zum Speichern der Daten verwendet, so wird dadurch immer auch die Ausfallsicherheit des Systems erhöht.

   wahr ○ falsch ⊗

8. Für Relation $R(AB)$ mit 100 Tupeln und Relation $S(\underline{A}C)$ mit 10 Tupeln enthält $\Pi_A(R) \ltimes \Pi_A(S)$ maximal 10 Tupel.

   wahr ⊗ falsch ○

9. Bei Cross-Joins spielt es keine Rolle, ob Projektionen vor oder nach dem Join ausgeführt werden.

   wahr ⊗ falsch ○

10. Die Größe der Zwischenergebnisse spielt bei Pipelining keine Rolle. Solange also kein Materializing-Schritt dazwischen kommt, ist die Join- Reihenfolge egal.

    wahr ○ falsch ⊗

(Pro korrekter Antwort 1.5 Punkte, **pro inkorrekter Antwort -1.5 Punkte**, pro nicht beantworteter Frage 0 Punkte, für die gesamte Aufgabe mindestens 0 Punkte)

Aufgabe 7: Embedded SQL/SQLJ (4)

Vervollständigen Sie folgendes Java Programm mit Embedded SQL Programm (in SQLJ) sodass die `matrNr`, der `name` und das `semester` auf der Konsole ausgegeben werden (`System.out.println`).

```
...
#sql iterator StudIterator(int matrNr,String name,int semester);
StudIterator iStud;
#sql iStud = { SELECT matrNr, name, semester FROM Studenten };
```

```
while(iStud.next()) {
  System.out.println(iStud.matrNr()+','
    +iStud.name()+','
    +iStud.semester());
}
```

```
iStud.close();
...
```

Aufgabe 8: Java (5)

Schreiben Sie ein Java Programm unter der Verwendung von JDBC, das eine Verbindung zu einer lokalen *Postgres Datenbank* öffnet und ein SQL Statement absetzt.

Um das Schließen der Verbindung und um eine Fehlerbehandlung brauchen Sie sich **nicht** kümmern.

Verbindung zur PG Datenbank öffnen (Server läuft auf `localhost`, Benutzername `exam`, Passwort `dbs`). [2]

```
Class.forName('org.postgresql.Driver');
Connection c = DriverManager.getConnection('jdbc:postgresql://localhost',
    'exam',
    'dbs');
** oder ***
DriverManager.registerDriver(new Driver());
PGSimpleDataSource ds = new PGSimpleDataSource();
ds.setServerName('localhost') ;
ds.setUser('exam') ;
ds.setPassword('dbs') ;
Connection c = ds.getConnection();
```

Erstellen Sie ein `Statement` und rufen Sie damit diese *stored procedure* auf: `proc` mit zwei Parametern: `'exam'` und 1. [3]

```
Statement s = c.createStatement();
s.executeQuery('{call proc('exam', 1) }');
```

Gesamtpunkte: 75