DB – Praktikum 14

Indexe & Transaktionen

Vorbemerkungen

Eine Transaktion ist eine logische Arbeitseinheit. Entweder muss die gesamte Arbeit dieser Einheit abgeschlossen werden oder gar nichts davon. Transaktionen sind mit den ACID-Prinzipien «verwoben» also Atomarität, Konsistenz, Isolation, und Dauerhaftigkeit. Diese Prinzipien können aber nicht «gratis» angeboten werden. Die Datenbank muss mittels Sperren und Mechanismen wie «Multiversion Concurrency Control» die Voraussetzungen schaffen, dass keine unerwünschten Phänomene auftreten. In diesem Praktikum wollen wir einerseits mit Indexen etwas vertrauter werden und andererseits den Mechanismus und die Konsequenzen von Transaktionen etwas näher beleuchten.

Wir arbeiten im Praktikum 14 mit dem Schema `postgres_air`. Dieses haben Sie bereits im Praktikum_01 bereitgestellt. Eine detaillierte Beschreibung zu diesem Schema finden Sie im Dokument `A_Praktikum_14_Schema.pdf`. Wenn Sie diese Datenbank noch nicht installiert haben können Sie alternativ die untenstehenden Aufgaben zu Transaktionen auch mit einer beliebigen anderen Datenbank sinngemäss umsetzen (z.B. der Bier-DB aus dem Praktikum). Ohne `postgres_air` machen aber die beiden Aufgaben 2 und 3 keinen Sinn (schauen Sie dann dort einfach zu). Alle unten angeführten SQL-Anweisungen finden Sie im Skript `S_Praktikum_14.sql`.

Nutzen Sie für die Aufgaben auch die offizielle PostgreSQL-Dokumentation.

Aufgabe 1: Auswirkung von Indexen und Primärschlüsseln

Um die Wirkung von Indexen zu „erfahren“, braucht es natürlich entsprechende Datenmengen. Unsere bisherigen Übungsdatenbanken waren dafür viel zu klein. Für die Aufgabe 1 müssen wir deshalb zuerst eine etwas grössere Tabelle erzeugen und füllen.

Erstellen Sie dazu im Schema `postgres_air` folgende Tabelle:

```
DROP TABLE IF EXISTS perftable;

CREATE TABLE perftable (one INT, two INT);
```

Studieren Sie anschliessend folgende Anweisung (ziehen Sie dazu wenn nötig die PostgreSQL-Dokumentation bei) und führen Sie sie aus:

```
INSERT INTO perftable
SELECT rnum + FLOOR(RANDOM()*10)::INT AS one,
       FLOOR(RANDOM()*100000000)::INT AS two
FROM (SELECT (num*10) AS rnum
      FROM GENERATE_SERIES (1,10000000) AS s(num)
      ORDER BY RANDOM()) AS x;
```

Führen Sie anschliessend folgenden Befehl aus: `SELECT * FROM perftable;` notieren Sie sich die Laufzeit. Überlegen Sie, wie die Daten dieser Tabelle gespeichert wurden und wie PostgreSQL für diese SELECT-Anweisung darauf zugreift.

Idealerweise sollte der Befehl eine Laufzeit im einstelligen Sekundenbereich haben. Ist das auf Ihrem Rechner nicht der Fall (abhängig von OS, Memory, etc.), dann passen Sie die Obergrenzen im INSERT-Kommando entsprechend an, um in diesen Bereich zu kommen.

Führen Sie nun folgende zwei Abfragen aus:

```
SELECT * FROM perftable WHERE one=8052;
```

```
SELECT * FROM perftable WHERE one BETWEEN 8000 AND 9000;
```

Die Abfragen liegen in derselben Grössenordnung der Laufzeit wie die Abfrage der ganzen Tabelle. Erklären Sie warum.

Erstellen Sie nun einen Index auf Attribut «one».

```
CREATE INDEX perfindex ON perftable(one);
```

Lassen Sie alle drei Abfragen nochmals laufen. Was passiert und warum?

Löschen Sie den Index mit `DROP INDEX perfindex;`

Fügen Sie dann mit ALTER TABLE der Tabelle einen Primärschlüssel hinzu:

```
ALTER TABLE perftable ADD PRIMARY KEY(one);
```

Lassen Sie alle drei Abfragen wiederum laufen. Was passiert jetzt und warum?

Aufgabe 2: Auswirkung von Indexen auf das Lesen

Erstellen Sie eine Liste aller Passagiere mit dem Namen ‚DANI WHEELER' (die Namen sind in dieser Tabelle alle gross geschrieben). Messen Sie zunächst die Laufzeit ohne Index und danach mit Index.

```
SELECT * FROM passenger
WHERE first_name = 'DANI' AND last_name = 'WHEELER';
```

Erzeugen Sie dann *einen* kombinierten Index auf den beiden Attributen `first_name` und `last_name`:

```
CREATE INDEX IX_Passenger ON passenger(first_name,last_name);
```

Vergleichen Sie die beiden Ausführungspläne und die Laufzeiten.

Löschen Sie dann diesen Index wieder.

```
DROP INDEX IX_Passenger;
```

Aufgabe 3: Auswirkung von Indexen auf das Schreiben

Für die Aufgabe 3 erstellen wir zuerst eine Kopie der Tabelle passenger:

```
CREATE TABLE passenger2 AS SELECT * FROM passenger;
```

Nun fügen wir 200000 Datensätze aus der Tabelle passenger in die neu erstellte Tabelle ein. Merken Sie sich die Laufzeit.

```
INSERT INTO passenger2 SELECT * FROM passenger LIMIT 200000;
```

Nun erstellen wir einen Index auf die Spalte booking_id:

```
CREATE INDEX IX_Passenger2_1 ON passenger2(booking_id);
```

Nun fügen wir nochmals 200000 Datensätze aus der Tabelle passenger in die neu erstellte Tabelle ein. Merken Sie sich auch hier die Laufzeit.

```
INSERT INTO passenger2 SELECT * FROM passenger LIMIT 200000;
```

Jetzt erstellen wir noch einen zweiten Index auf dem Attribut first_name:

```
CREATE INDEX IX_Passenger2_2 ON passenger2(first_name);
```

Nun fügen wir nochmals 200000 Datensätze aus der Tabelle title in die neu erstellte Tabelle ein. Merken Sie sich auch hier die Laufzeit.

```
INSERT INTO passenger2 SELECT * FROM passenger LIMIT 200000;
```

Der Aufbau der Indexe erhöht die Einfügezeit von 200K Datensätzen signifikant. D.h. obwohl der Index die Abfragezeit reduziert, wird die Einfügezeit dadurch erhöht, dass die Indexe dynamisch angepasst werden müssen. Wenn es jedoch viele Abfrage- und wenige Einfügeoperationen gibt, lohnt sich der Index – andernfalls eventuell nicht. In der Praxis müssen daher immer die Laufzeiten aller Abfragen analysiert werden, um zu entscheiden, ob ein Index sinnvoll ist.

Aufgabe 4: «My first transaction»

Transaktionen werden mit `BEGIN TRANSACTION;` eingeleitet (in PostgreSQL reicht auch nur `BEGIN` und `END`) und mit `COMMIT;` oder `ROLLBACK;` abgeschlossen. Wenn diese Anweisungen weggelassen werden so arbeitet man im so genannten „Autocommit-Modus“, d.h. dann wird jede einzelne Anweisung als Transaktion behandelt.

Führen Sie zunächst den folgenden SQL-Befehl aus:

```
SELECT * FROM aircraft WHERE model LIKE 'Boeing%';
```

Sie sollten eine Resultattabelle mit drei Zeilen erhalten.

Wenn wir die Abfrage nun mit

```
BEGIN TRANSACTION;
SELECT * FROM aircraft WHERE model LIKE 'Boeing%';
COMMIT;
```

kapseln, ändert sich vorerst am Resultat nichts. Das liegt daran, dass wir nur lesen, und dass wir alleine mit der Datenbank arbeiten. Wir werden das im Folgenden ändern.

Explizite Transaktionen werden in der Regel in Anwendungen bzw. in gespeicherten Prozeduren und Triggern eingesetzt.

Aufgabe 5: Einmal «hin und zurück»

Idealerweise läuft eine Transaktion erfolgreich durch. Da ja, wie erwähnt, die Transaktion „ganz oder gar nicht" ausgeführt werden soll, kann die Datenbank Änderungen einer Transaktion erst dann persistent machen, wenn der Gesamtablauf erfolgreich war. Man sagt dann, die Änderungen werden „committed".

Wenn wir den „ganz oder gar nicht"-Effekt sehen wollen, ist es aber fast einfacher, ein „gar nicht" zu provozieren. Damit haben wir unmittelbar ein anderes Verhalten als ohne Transaktionsmechanismus.

Führen Sie folgende Anweisungen einzeln aus:

```
ALTER TABLE flight DROP CONSTRAINT aircraft_code_fk;

SELECT * FROM aircraft;

BEGIN TRANSACTION;
DELETE FROM aircraft WHERE model LIKE 'Boeing%';
SELECT * FROM aircraft;
ROLLBACK;

SELECT * FROM aircraft;
```

Was ist passiert? Warum?

Aufgabe 6: Verschiedene Stufen der Isolation

Versuchen Sie, nachzuvollziehen, inwiefern die beiden ISOLATION LEVELs `READ COMMITTED` und `REPEATABLE READ` zu unterschiedlichem Verhalten führen. Dazu brauchen wir nun zwei Sessions.

Schauen wir uns zuerst den Eintrag für den Flughafen Zürich an:

```
SELECT * FROM airport WHERE airport_code = 'ZRH';
```

Beginnen Sie nun in einer ersten „Session" folgende Transaktion (vorerst nur erste zwei Anweisungen):

```
BEGIN TRANSACTION;
UPDATE airport SET airport_name = 'Zürich Airport'
WHERE airport_code = 'ZRH';
COMMIT;
```

Führen Sie dann in einer zweiten Session folgende Transaktion - schrittweise - aus:

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM airport WHERE airport_code = 'ZRH';
SELECT * FROM airport WHERE airport_code = 'ZRH';
COMMIT;
```

Schliessen Sie die erste Transaktion ab mit commit. Wiederholen Sie dann den ganzen Vorgang, aber diesmal mit dem Isolationslevel REPEATABLE READ.

Das zeigt den Unterschied zwischen den beiden Isolation Levels, d.h., wenn Sie nach Abschluss der zweiten Transaktion in der ersten Transaktion nochmals das gleiche SELECT-Statement ausführen, dann sollte der Wert sich ändern (READ COMMITTED), resp. gleich bleiben (REPEATABLE READ).

Der gleiche Effekt wie bei REPEATABLE READ zeigt sich auch beim Isolationslevel SERIALIZABLE.

Aufgabe 7: «My first deadlock»

Erzwingen Sie einen Deadlock mit Hilfe der beiden Tabellen `airport` und `passenger`.

Transaktion 1:

```
BEGIN TRANSACTION;
UPDATE airport SET airport_name = 'Malaga Airport'
WHERE airport_code = 'AGP';
UPDATE passenger SET first_name = 'DANY'
WHERE passenger_id = 8583106;
COMMIT;
```

Transaktion 2:

```
BEGIN TRANSACTION;
UPDATE passenger SET first_name = 'DANY'
WHERE passenger_id = 8583106;
UPDATE airport SET airport_name = 'Malaga Airport'
WHERE airport_code = 'AGP';
COMMIT;
```

Führen Sie die beiden Transaktionen schrittweise alternierend aus. Beobachten Sie, was PostgreSQL als Konsequenz gegen den Deadlock unternimmt. Überlegen Sie sich, ob dieses Vorgehen „fair“ ist. Was müssen Sie im Anschluss an diese Form der Fehlerbehandlung unternehmen?

Nebenbemerkung: Beachten Sie auch die Bemerkung im letzten Satz des ersten Absatzes unter 13.3.4 in der PostgreSQL-Dokumentation (https://www.postgresql.org/docs/15/explicit-locking.html)

Aufgabe 8: Misslungene Serialisierung

Ein Deadlock ist, wie Sie in Aufgabe 7 gesehen haben, «fatal»: es ist unmöglich, beide Transaktionen zu einem guten Ende zu bringen. Dies unrühmliche Ende – ein erzwungener Abort – kann eine Transaktion aber auch ohne Deadlock ereilen. Dies ist z.B. dann der Fall, wenn zwei Transaktionen widersprüchliche Modifikationen an denselben Tupeln vornehmen wollen.

Versuchen Sie ein solches Verhalten zu provozieren. Starten Sie eine erste Transaktion, und modifizieren Sie ein spezifisches Tupel aus `passenger`:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM passenger WHERE passenger_id = 8583106;
UPDATE passenger SET first_name = 'DANY'
WHERE passenger_id = 8583106;
COMMIT;
```

Starten Sie dann eine zweite Transaktion, und machen Sie dasselbe:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM passenger WHERE passenger_id = 8583106;
UPDATE passenger SET first_name = 'DANI'
WHERE passenger_id = 8583106;
COMMIT;
```

Was ist der Wert, den Sie sehen? Wie erklären Sie sich das Verhalten bis zu diesem Punkt? Schreiben Sie nun aus der zweiten Transaktion auf das gleiche Tupel. Was passiert?