UE (GK) - Optimierung von Datenbankabfragen

Voraussetzungen

Starte deine Postgres-Instanz aus den letzten Aufgabenstellungen und lege eine leere Datenbank an. Anbei findest du eine Webshop-Testdatenbank mit einigen Beispieldaten (10000 Datensaetze pro Tabelle). Importiere diese Daten zunaechst mit

```
webshop2=# \i webshop.sql
webshop2=# \i webshop_data.sql
```

Du kannst dir mittels `\d` die Tabellennamen anzeigen lassen und zB mittels `\d kunde` die Definition der Kundentabelle.

Aufgabenstellung

In der Datenbankanwendung werden die folgenden Queries haeufig aufgefuehrt. Dafuer wurden auch Indexes angelegt. Untersuche mittels EXPLAIN bzw EXPLAIN ANALYZE [1], ob die folgenden Queries den Index effizient nutzen koennen, bzw. ob man sie durch anderes Setzen von Indexen besser beschleunigen kann und beantworte die Fragestellungen. Die Webseite [2] liefert dazu gute Erklaerungen und Tipps.

Beispiel 1:

Query

```
CREATE INDEX idx_bdatum ON bestellung (bdatum);

SELECT bnr, bstatus, bdatum
    FROM bestellung
    WHERE TO_CHAR(bdatum, 'YYYY') = '2017';
```

Frage

Kann die gegebene Query den Index nutzen? Begruende deine Antwort! Falls nein, wie kann man die Query entsprechend aendern? Kann man vielleicht alternativ auch den Index aendern?

Analyse

Die Funktion `TO_CHAR(bdatum, 'YYYY')` wandelt das Datum in einen String um, wodurch der Index auf `bdatum` ineffektiv wird, da die Datenbank die Transformation auf jedem Datensatz ausführen muss und somit nicht direkt auf den Index zugreifen kann.

Optimierung

```
-- Das genaue Jahr extrahieren
CREATE INDEX idx_bdatum_year ON bestellung (EXTRACT(YEAR FROM bdatum));

EXPLAIN ANALYSE SELECT bnr, bstatus, bdatum
    FROM bestellung
    WHERE EXTRACT(YEAR FROM bdatum) = 2017;

-- Alternative
EXPLAIN ANALYSE SELECT bnr, bstatus, bdatum
    FROM bestellung
    WHERE bdatum >= to_date(2017::varchar, 'YYYY') AND bdatum <
to_date(2018::varchar, 'YYYY');

-- Alternative
EXPLAIN ANALYSE SELECT bnr, bstatus, bdatum
    FROM bestellung
    WHERE bdatum = to_date(2017::varchar, 'YYYY');
```

Beispiel 2:

Query

```
CREATE INDEX idx_zusammen ON bestellung (bstatus, bdatum);

SELECT id, bstatus, bdatum
    FROM bestellung
    WHERE bstatus = %
    ORDER BY bdatum DESC
    LIMIT 1;
```

`%` replace with a specific value

Frage

Was liefert die gegebene Query als Resultat? Kann sie den Index effizient nutzen? Falls nein, wie muesste man den Index dafuer aendern?

Analyse

Die Query sucht nach einem spezifischen `bstatus` und dem neuesten `bdatum`. Der Index sollte effektiv genutzt werden, da zuerst nach `bstatus` gefiltert wird und anschließend `bdatum` in absteigender Reihenfolge sortiert wird.

Optimierung

```
-- Create Temporary Column
ALTER TABLE bestellung ADD COLUMN bstatus_short TEXT;
```

```
-- Extract Numbers from bstatus
UPDATE bestellung
SET bstatus_short = SUBSTRING(bstatus FROM 'bstatus_([0-9]+)');

-- Original Column löschen und neue Column umbenennen
ALTER TABLE bestellung DROP COLUMN bstatus;
ALTER TABLE bestellung RENAME COLUMN bstatus_short TO bstatus;

-- Or Rename Columns
ALTER TABLE bestellung RENAME COLUMN bstatus TO bstatus_old;
ALTER TABLE bestellung RENAME COLUMN bstatus_short TO bstatus;

-- DESC hinzufügen um ein re-sorting der Daten zu umgehen
CREATE INDEX idx_zusammen_desc ON bestellung_id (bstatus, bdatum DESC);

-- Index mit allen Spalten erstellen
CREATE INDEX idx_zusammen_covering ON bestellung_id (bstatus, bdatum DESC) INCLUDE
(id);
```

Beispiel 3:

Query

```
CREATE INDEX idx_artikel ON artikel (anr, vstueckz);

SELECT id, anr, vstueckz
    FROM artikel
    WHERE anr = %
    AND vstueckz = %;

SELECT id, anr, vstueckz
    FROM artikel
    WHERE vstueckz = %;
```

`%` replace with a specific value

Frage

Koennen die gegebenen Queries den Index effizient nutzen? Begruende deine Antwort. Falls nein, wie muesste man ihn Index dafuer aendern?

Analyse

- Die erste Query kann den Index effektiv nutzen, da sie sowohl `anr` als auch `vstueckz` nutzt.
- Die zweite Query nutzt nur `vstueckz`, und der Index ist nicht optimal, da `anr` der führende Index-Schlüssel ist.

Optimierung

```
-- Index fuer die zweite Query
CREATE INDEX idx_vstueckz ON artikel (vstueckz);
```

Beispiel 4:

Query

```
CREATE INDEX idx_emails ON kunde (email varchar_pattern_ops);

SELECT id from kunde where email like 'rlugner%';
SELECT id from kunde where email like '%moertel.at';
```

Frage

Unterscheiden sich die beiden Queries stark in ihrer Performance? Falls ja, warum? Weiters, falls ja, wie sollte man Queries und Index aendern, damit fuer beide der Index genutzt werden kann?

Analyse

- Die erste Query kann den Index effektiv nutzen, da das Suchmuster mit einem festen Präfix beginnt.
- Die zweite Query kann den Index nicht nutzen, da sie ein Suffix sucht, was nicht direkt unterstützt wird.

Optimierung

```
-- Umgekehrter Index fuer Suffix-Suche
CREATE INDEX idx_emails_reverse ON kunde (reverse(email));

-- Umgekehrter Index fuer Suffix-Suche mit varchar_pattern_ops
CREATE INDEX idx_emails_reverse_suffix ON kunde (reverse(email)
varchar_pattern_ops);
```

(Optimierung nicht möglich)

Beispiel 5:

Query

```
CREATE INDEX idx_bestellung_2 ON bestellung (bdatum, kunde_id);

SELECT id, bdatum, kunde_id
    FROM bestellung
    WHERE bdatum > CURRENT_DATE - INTERVAL '30' YEAR
    AND kunde_id = 42;
```

Frage

Kann die angegebene Query den Index so effizient wie moeglich nutzen? Falls nein, wie muesste man ihn dafuer aendern?

Analyse

Die Query sollte den Index effektiv nutzen, da beide Bedingungen (Datum und Kunde ID) im Index enthalten sind.

Optimierung

```
-- Id als führenden Index-Schlüssel setzen
CREATE INDEX idx_bestellung_id ON bestellung (kunde_id, bdatum);

-- Index mit allen Spalten erstellen
CREATE INDEX idx_bestellung_covering ON bestellung (kunde_id, bdatum) INCLUDE
(id);
```

(Optimierung nicht möglich)

Beispiel 6:

Frage

Erklaere, wie und warum sich durch die Verwendung von Prepared Statements die Performance einer Query unter Umstaenden auch stark verschlechtern kann.

Hint: Betrachte zB die folgende Query:

```
SELECT id from bestellung where bdatum > '2017-01-01' and bdatum <= '2017-01-02'
```

Antwort

Prepared statements ist ein Feature, dass einen ermöglicht die selbe (oder ähnliche) Query mehrmals mit hoher Effizienz auszuführen.

1. **Prepare:** Ein Query template wird erstellt und an die Datenbank gesendet. Dabei werden Parameter verwendet, die später durch Werte ersetzt werden.

Die Datenbank kompiliert, analysiert und schreibt den Query um und erstellt einen Ausführungsplan. Dieser Plan wird dann als Serverseitiges Objekt gespeichert.

1. **Execute:** Dabei werden im prepared statement Parameter durch Werte ersetzt. Daraufhin wird das ganze geplant und ausgeführt. Dies umgeht die Notwendigkeit den Query jedes mal neu zu kompilieren, analysieren und optimieren.

```
-- Example
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

Nach dem Beenden einer Session werden die prepared statements alle wieder verworfen. Daher muss es jedes mal neu erstellt werden. Das bedeutet auch, dass ein prepared statement nicht von mehreren database clients gleichzeitig verwendet werden kann. Manuell können pepared statements mit dem Befehl `DEALLOCATE` gelöscht werden.

Prepared Statements können die Leistung einer Datenbankabfrage auch verschlechtern. In manchen Situationen wird ein allgemeiner Ausführungsplan erstellt, welcher für alle mögliche Parameterwerte angepasst wird und funktionieren muss. Dies kann bei sepzifischen Datenverteilungen suboptimal sein.

Beispiel: Anhand des Beispiels könnte ein generischer Plan ineffizient werden, wenn beispielsweise bestimmte Zeiträume besonders viele Daten enthalten. Der Plan ist nicht speziell für diese Bedingungen optimiert.

- **Generischer Plan**: Nicht optimal für spezifische Datenverteilungen oder ungewöhnliche Bedingungen.
- **Ineffiziente Indexnutzung**: Bei datenintensiven Zeiträumen könnte der Plan ineffiziente Zugriffsmethoden wie Index Scans über zu viele Datenpunkte nutzen, anstatt effizientere Vollscans zu verwenden.

Generische Ausführungspläne von Prepared Statements sind nicht immer die beste Wahl, besonders wenn die Datenverteilung ungleich ist oder sich häufig ändert. In solchen Fällen könnte ein dynamisch erstellter Plan direkt vom DBMS effizienter sein.

Probleme

Datenbank importieren

Error beim ausfuehren von `webshop_data.sql`:

```
[2024-04-13 00:18:33] [57014] ERROR: COPY from stdin failed: COPY commands are
only supported using the CopyManager API.
[2024-04-13 00:18:33] Wobei: COPY bestellartikel, line 1
```

Ausfuehren im Docker Container

```
# Copy file to container
docker cp D:\Path\To\webshop_data.sql dev-postgres:/webshop_data.sql
# Execute file in container
docker exec -it dev-postgres psql -U postgres -d webshop2 -f /webshop_data.sql
```

Abgabe

Fasse deine Ausarbeitungen und Antworten in einem Protokoll zusammen, gib dieses ab, und melde dich fuer ein Abgabegespraech an.

Quellen:

[1] https://www.postgresql.org/docs/current/using-explain.html

[2] https://use-the-index-luke.com/sql/where-clause

[3] webshop.zip

[4] https://www.postgresql.org/docs/current/sql-prepare.html