

16.01.2024

Übungsblatt 11 – Lösungsvorschlag

Diskussionsteil (im PS zu lösen; keine Abgabe nötig)

- a) ☐ ★ Diskutieren Sie, ob ein Datenbanksystem für alle möglicherweise auftretenden Anfragen optimiert werden kann und begründen Sie Ihre Antwort. Falls nein: bezüglich welcher Abfragen sollte ein Datenbanksystem optimiert werden? Wie finden Sie diese Abfragen?

Lösung



Ein Datenbanksystem kann nicht auf jede möglicherweise auftretende Anfrage optimiert werden. So können z.B. nicht lesende und schreibende Abfragen meist nicht gleich stark unterstützt werden, da sich die Anforderungen an Indexstrukturen etc. stark unterscheiden. Hier wird also meist auf eine Query-Art optimiert oder versucht, ein passendes Gleichgewicht zu finden. Ein Datenbanksystem sollte prinzipiell immer auf Performance-kritische Queries optimiert werden. Dies sind meist Queries, die oft ausgeführt werden. Diese Abfragen kann man auf verschiedene Weise aufspüren: einerseits im Gespräch mit den Anwendungsentwicklern und den (direkten) Datenbank-Nutzern und andererseits über Datenbank-Logging. Beim Logging kann die Datenbank so konfiguriert werden, dass alle Queries oder auch nur jene Queries die z.B. länger als eine konfigurierbare Sekundenanzahl zur Berechnung benötigen, geloggt werden. Dieses Log kann dann dazu verwendet werden, die kritischen und häufigsten Abfragen und Abfragearten festzustellen und das System dahingehend zu optimieren.

- b) ☐ ★★ Man spricht oft davon, dass eine Query direkt “aus dem Index beantwortet werden kann” (“index-only”). Was bedeutet das für die Query-Abarbeitung und für die Performance der Datenbank? Welche Bedingungen müssen dafür erfüllt sein?

Lösung



Eine Query kann aus dem Index beantwortet werden, wenn zur Berechnung des Ergebnisses nur Daten direkt aus den Indexstrukturen (die bestenfalls im RAM liegen) benötigt werden. Entsprechend benötigt die Berechnung keinen Zugriff auf Hintergrundspeicher über den Datenpointer und kann somit performanter abgearbeitet werden. Die Bedingung für eine Performancesteigerung ist, dass für die Bearbeitung der Abfrage nur Daten herangezogen werden müssen, die in einer passenden Indexstruktur enthalten sind. Hier gehen wir davon aus, dass die Indexstruktur bereits im RAM liegt.

- c) ☐ ★★★ Um Datenbanken sinnvoll zu nutzen, müssen wir oft von Applikationen aus auf eine Datenbank zugreifen, um Daten entweder auszulesen oder in die Datenbank zu schreiben. Beantworten Sie dazu folgende Fragen:

- Was versteht man unter dem Cursor-Prinzip? Wofür wird es verwendet?

Lösung




Cursor erlauben es uns, Tupel für Tupel über eine Ergebnismenge zu iterieren. Dies benötigen wir, um in Programmiersprachen, die nicht mengenorientiert sind, auf Ergebnisse einer SQL-Abfrage zugreifen zu können.

- Was sind ODBC und JDBC?

Lösung



Bei diesen beiden Technologien handelt es sich um Schnittstellen zwischen Anwendungsprogrammen und Datenbanksysteme. Sie ermöglichen es einem Programm, mit einer Datenbank zu kommunizieren und damit zum Beispiel Abfragen abzusetzen.

- d)  Zur Vorbereitung auf einen Teil der Hausaufgabe, wollen wir von einer Applikation aus auf die `pagila` Datenbank zugreifen und Daten auslesen. Schreiben Sie zu diesem Zwecke eine kleine Applikation, die sich mit der Datenbank verbindet, die Spalten `title` und `rental_rate` aus der Tabelle `film` ausliest und das Ergebnis auf der Konsole ausgibt. Berechnen Sie in Ihrem Programm weiters die Summe über die Spalte `rental_rate` und geben Sie diese nach den Zeilen der Tabelle aus.

Hinweis



Starten Sie, je nach Wahl Ihrer Programmiersprache, mit den folgenden Dokumentationsseiten:

- C/C++: <https://www.postgresql.org/docs/13/libpq.html>
- Java: <https://jdbc.postgresql.org/documentation/>
- Python: <https://www.psycopg.org/docs/>

Lösung



Siehe `demo.c`, bzw. `demo.py`

Hausaufgabenteil (Zuhause zu lösen; Abgabe nötig)

Aufgabe 1 (Tooling)

[3 Punkte]

Ziel dieser Aufgabe ist es Ihnen zu zeigen, welche Tools PostgreSQL für Datenbanktuning und zur Lösung von Performanceproblemen bereitstellt. Die hier behandelten Tools gehören allesamt zum sogenannten `Statistics Collector` von PostgreSQL. Beginnen Sie daher, indem Sie sich die Dokumentation¹ dazu durchlesen.

¹<https://www.postgresql.org/docs/15/monitoring-stats.html>

Hinweis



Stellen Sie sicher, dass Ihr System so konfiguriert ist, dass Statistiken gesammelt werden. Im Artikel [Settings Parameters^a](https://www.postgresql.org/docs/15/config-setting.html) finden Sie Informationen darüber, wie Sie die Einstellungen Ihrer PostgreSQL Datenbank ändern können.

^a<https://www.postgresql.org/docs/15/config-setting.html>

- a) **1.5 Punkte** Sie administrieren die Datenbank eines Buchhaltungssystems und bei jeder Operation, die Buchungen aus der Datenbank ausliest bzw. Buchungen in diese schreiben soll, "hängt" die Datenbank und damit auch das Buchhaltungssystem. Andere Operationen, wie zum Beispiel das Auslesen von Kundendaten, laufen einwandfrei.

Sehen Sie sich die View `pg_stat_activity` an und beantworten Sie die folgenden Fragen.

- Was könnte hier die Ursache sein?
- Welche Informationen können Sie aus der Ausgabe dieser View herauslesen?
- Können Sie diese View verwenden, um das Problem zu lösen? Wie würden Sie vorgehen?

Abgabe



1a.pdf

Lösung



Die wahrscheinliche Ursache hier ist, dass eine sehr langsame Transaktion eine Sperre auf die Buchungstabelle hält und alle späteren Operationen dadurch warten bzw. irgendwann in ein Timeout laufen.

Um das zu überprüfen und die langsame Transaktion zu identifizieren, können Sie die `pg_stat_activity`-View benutzen. Hier sollten, falls die Ursache tatsächlich eine langsame Transaktion ist und im System einige nicht reagierende Transaktionen laufen (die "hängenden" Prozesse der Anwender) einige Zeilen zurückbekommen, die warten (zu erkennen am Inhalt der Spalten `wait_event_type` und `wait_event`).

Auch die langsame Transaktion können Sie mit dieser View identifizieren. Die Spalte `xact_start` sagt Ihnen etwa, wann eine Transaktion begonnen hat.

- b) **1.5 Punkte** Sie werden beauftragt, die aktuelle Indizierungsstrategie für eine Datenbank zu evaluieren. Sie sollen ermitteln, ob die vorhandenen Indizes ihren Zweck erfüllen, ob einige davon eventuell gelöscht werden könnten und ob neue Indizes angelegt werden sollten.

Welche Views, die Ihnen der Statistics Collector zur Verfügung stellt, können Sie hier verwenden? Wie sieht Ihr weiteres Vorgehen aus?

Abgabe



1b.pdf

Lösung



Hier sind drei Views besonders nützlich: `pg_stat_user_indexes`, `pg_statio_user_indexes` und `pg_stat_user_tables`.

Die ersten beiden Views sagen Ihnen, inwiefern vorhandene Indizes wirklich genutzt werden. Mit `pg_stat_user_indexes` können Sie herausfinden, wie oft ein Index insgesamt benutzt wurde (`idx_scan`) und wie viele Zeilen dadurch ausgelesen wurden (`idx_tup_read` bzw. `idx_tup_fetch`). Mithilfe von `pg_statio_user_indexes` können Sie sich ein Bild davon verschaffen, was das für Zugriffe auf den Hintergrundspeicher bedeutet.

Kandidaten für Indizes, die eventuell gelöscht werden können, sind dann solche, die kaum benutzt werden.

Um festzustellen, ob Indizes fehlen, können Sie sich die View `pg_stat_user_tables` ansehen. Diese enthält eine Zeile pro Tabelle in der Datenbank. Die Spalten `seq_scan` und `idx_scan` sagen Ihnen, wie oft auf diese Tabelle über sequentielle Scans bzw. über Indizes zugegriffen wurden. Ist die Anzahl der sequentiellen Zugriffe hier hoch und die Anzahl der Index-Zugriffe niedrig (oder gar 0), ist das ein Hinweis dafür, dass Indizes auf diese Tabelle fehlen könnten.

Aufgabe 2 (Query-Tuning)

[3 Punkte]

In dieser Aufgabe werden Sie verschiedene Möglichkeiten anwenden, eine gegebene Abfrage zu optimieren. Wir werden uns dabei auf zwei Möglichkeiten konzentrieren:

- Optimieren einer Abfrage, indem wir sie anders formulieren (Stichwort Rewriting).
- Optimieren einer Abfrage, indem wir die physische Datenstruktur der Datenbank so ändern bzw. erweitern, dass die Abfrage schneller beantwortet werden kann (Stichwort Indizes).

Wir werden bei den folgenden Aufgaben zum Teil mit der `pagila` Datenbank arbeiten. Da in dieser Datenbank allerdings schon recht viele Indizes definiert sind, müssen Sie diese zuerst löschen, damit Sie die folgenden Aufgaben richtig bearbeiten können. Löschen Sie zu zuerst die Indizes in Ihrer `pagila` Datenbank (siehe `drop_indices.sqlOLAT`) und bearbeiten Sie dann die folgenden Aufgaben.



a) 1.5 Punkte Gegeben sei die folgende Abfrage (auf der bekannten `pagila` Datenbank):

```
1  SELECT first_name, last_name, count(film_id)
2  FROM film_actor
3  INNER JOIN actor
4  ON actor.actor_id = film_actor.actor_id
5  WHERE film_id != ALL(SELECT film_id FROM film WHERE length < 120)
6  GROUP BY first_name, last_name
7  ORDER BY first_name, last_name
```

Diese Abfrage wird von PostgreSQL relativ ineffizient abgearbeitet. Finden Sie eine äquivalente Abfrage, welche das gleiche Ergebnis liefert und effizienter ausgeführt wird. Messen Sie mithilfe Ihres Client-Tools (etwa `pgAdmin`), wie lange beide Varianten für die Ausführung benötigen. Sehen Sie sich weiters die Ausführungspläne der beiden Varianten an und versuchen Sie zu erklären, warum die eine Variante schneller ist als die andere. Schreiben Sie Ihre Überlegungen und alle zugehörigen Materialien (z.B. die Ausführungspläne) in einem PDF-Dokument zusammen.

Abgabe



 2a_faster.sql
 2a_explanations.pdf

Lösung



```
1  SELECT first_name, last_name, count(film_id)
2  FROM film_actor
3  INNER JOIN actor
4  ON actor.actor_id = film_actor.actor_id
5  WHERE film_id NOT IN (SELECT film_id FROM film WHERE length < 120)
6  GROUP BY first_name, last_name
7  ORDER BY first_name
```

PostgreSQL bereitet im Optimierungsvorgang einen gehashten Subplan vor, der die Ausführungszeit verkürzt. Dies gilt aber nur in diesem speziellen Fall, da die Subquery nur eine geringe Anzahl an Zeilen zurück gibt. Angenommen die Subquery returniert mehrere 1000 Zeilen, dann wäre dieser Ausführungsplan nicht mehr performant, da das Ergebnis des Subplans nun materialisiert wird. Daraus folgt, dass die Verwendung des IN Statements nur Vorteile bringt, wenn der Subplan weniger Zeilen auswählt.

Folgende Queries liefern (auf unserem Testsystem) ähnliche Performances wie die ursprüngliche Lösung:

```
1  SELECT first_name, last_name, count(film_id)
2  FROM film_actor
3  INNER JOIN actor
4  ON actor.actor_id = film_actor.actor_id
5  WHERE NOT EXISTS
6    (SELECT film_id FROM film WHERE length < 120
7      AND film.film_id = film_actor.film_id)
8  GROUP BY first_name, last_name
9  ORDER BY first_name, last_name
```

oder

```
1  SELECT first_name, last_name, count(film_actor.film_id)
2  FROM film_actor
3  INNER JOIN actor
4  ON actor.actor_id = film_actor.actor_id
5  LEFT JOIN film ON film_actor.film_id = film.film_id
6    AND length < 120
7  WHERE film.film_id IS NULL
8  GROUP BY first_name, last_name
9  ORDER BY first_name, last_name
```

- b) **1.5 Punkte** Gegeben sei die folgende Abfrage. Diese müssen Sie vorerst nicht ausführen — wir machen erst eine theoretische Betrachtung.

```
1  SELECT      customer_id,
2              last_name,
3              first_name
4  FROM        customer
5  ORDER BY    last_name ASC
```

Nehmen Sie an, die Tabelle `customer` enthält 1.000.000 Zeilen. Nehmen Sie weiters an, dass auf der Tabelle keine Indizes definiert sind. Welche(n) Index/Indizes können Sie definieren, damit die Ausführung dieser Query eventuell beschleunigt wird?

Führen Sie nun die Abfrage auf der `pagila` Datenbank aus, messen Sie, wie lang die Ausführung dauert, und sehen Sie sich den Ausführungsplan an (Speichern nicht vergessen — der Ausführungsplan muss in der Abgabe enthalten sein!). Erstellen Sie anschließend in der `Pagila`-Datenbank den Index bzw. die Indizes, welche(n) Sie vorhin entworfen haben. Starten Sie PostgreSQL neu (um eventuelle Einflüsse durch Caching zu vermeiden) und führen Sie die Abfrage erneut aus. Messen Sie die benötigte Zeit und speichern Sie den Ausführungsplan. Gibt es einen Unterschied? Wurde(n) der Index/die Indizes verwendet? Wenn ja, wie? Wenn nein, welchen Grund könnte das Ihrer Meinung nach haben? Recherchieren Sie und schreiben Sie Ihre Überlegungen und alle zugehörigen Materialien in einem PDF-Dokument zusammen.

Abgabe



2b_index.sql
 2b_explanations.pdf

Lösung



Für die gegebene Abfrage bietet sich ein sogenannter *covering index* an. Diesen erstellen wir zum Beispiel wie folgt:

```
1  CREATE INDEX idx_customer_lastname_firstname_id
2  ON      customer(
3      last_name,
4      first_name,
5      customer_id
6  )
```

Wir indizieren hier `last_name` als erstes, weil damit eventuell die Sortierungsoperation eingespart werden kann (das hängt aber davon ab, wie genau das Datenbanksystem die Operation umsetzt).

Wenn wir all das jetzt auf der `pagila` Datenbank ausführen, werden wir wahrscheinlich feststellen, dass PostgreSQL diesen Index ignoriert. Dies liegt daran, dass der Optimierer von PostgreSQL zu dem Schluss kommt, dass sich die Verwendung des Indexes nicht lohnt. In diese Entscheidung fließen mehrere Faktoren ein:

- Die Datenmengen in der `pagila` Datenbank sind recht klein. Dadurch sind *sequential scans* vergleichsweise günstig, da nicht viele Pages gelesen werden müssen und die Datenbank dadurch vergleichsweise einfach im RAM Platz hat.

- Für sogenannte *index only scans* ist aufgrund der Art und Weise, wie die physische Speicherorganisation von PostgreSQL funktioniert, ein gewisser Overhead erforderlich. *Index only scans* können "schief gehen", wodurch doch wieder die Daten der eigentlichen Tabelle gelesen werden müssen. Bevor PostgreSQL weiß, ob dies notwendig ist, muss es in der sogenannten *visibility map* nachsehen. Diese Operation verursacht einen Overhead.

Aufgabe 3 (Datenbankanbindung)

[4 Punkte]

In dieser Aufgabe werden Sie, ähnlich wie in der Diskussionsaufgabe, eine kleine Anwendung schreiben, welche mit der *pagila* Datenbank kommunizieren soll. Für die Wahl der Programmiersprache und Bibliothek, welche Sie für diese Aufgabe verwenden, stellen wir Ihnen folgende Optionen frei:

- C/C++ (mit *libpq*)
- Java (mit *pgJDBC*)
- Python (mit *Psycopg*)

Wählen Sie aus diesen Sprachen jene aus, mit der Sie am meisten Erfahrung haben. Sie finden im OLAT je eine Vorlage (*exercise3.c*, *exercise3.py*, *Exercise3.java*). Diese Vorlagen enthalten ein einfaches Konsolen-Interface, über welches Berichte aus der *Pagila*-Datenbank abgerufen werden können sollen.

- 0.5 Punkte** Implementieren Sie die Verbindungsverwaltung für den Zugriff auf die Datenbank. Achten Sie hier besonders auf korrekte Fehlerbehandlung und darauf, dass die Anwendung sinnvolle Fehlermeldungen ausgibt, wenn beim Verbindungsaufbau etwas schiefgeht.
- 1.5 Punkte** Implementieren Sie den Befehl `rental_report`, mit welchem sich ein Bericht über die erfolgten Entleihungen in einem frei wählbaren Zeitraum angezeigt werden soll. Verwenden Sie hierfür die SQL-Abfrage aus der Datei `3_query.sql` - der Bericht gibt aufgeschlüsselt nach Kategorie spaltenweise die Anzahl der entliehenen Filme pro Wochentag aus.

Der Datumsfilter für den Bericht muss über die Kommandozeile eingelesen werden. Ein Beispiel für die Verwendung dieses Befehls könnte etwa wie folgt aussehen:

```
(Pagila)> rental_report
From: 2005-06-01
To: 2005-06-30
```

category_name	mon	tue	wed	thu	fri	sat	sun	total
Action	30	38	75	74	73	83	91	464
Animation	34	46	89	85	82	81	72	489
Children	23	34	68	57	80	64	80	406
Classics	19	39	53	54	81	80	58	384
Comedy	37	30	58	61	75	69	53	383
Documentary	30	33	81	69	68	65	83	429
Drama	36	31	79	73	84	87	73	463
Family	27	41	88	65	62	83	94	460
Foreign	33	35	73	68	83	70	70	432
Games	25	30	67	77	59	67	67	392

Horror		22		27		62		71		56		61		67		366	
Music		18		36		66		56		64		56		52		348	
New		28		36		64		68		66		60		67		389	
Sci-Fi		41		30		76		77		73		73		92		462	
Sports		28		39		99		71		89		81		90		497	
Travel		30		30		55		55		58		67		50		345	

Achten Sie auch hier wieder auf eine sinnvolle Fehlerbehandlung.

- c) 2 Punkte In der vorherigen Unteraufgabe wurde eine komplexere Abfrage direkt im Programmcode verwendet. Eine Alternative dazu ist es, in der Datenbank eine tabellenwertige Funktion zu erstellen und diese aufzurufen, um das Resultat zu erhalten. Sehen Sie sich hierzu die PostgreSQL-Dokumentation für benutzerdefinierte Funktionen² an, im Speziellen die Funktionen mit einem RETURNS TABLE Ausdruck.

Legen Sie dann in Ihrer Pagila-Datenbank eine Funktion an, welche denselben Bericht wie in Unteraufgabe b) generiert (die Funktion muss also auch einen Datumsbereich als Parameter entgegennehmen!). Implementieren Sie anschließend in Ihrer Applikation den Befehl `rental_report_function`. Dieser Befehl soll die Query nicht direkt wie in Unteraufgabe b) übergeben, der Aufruf soll folgende Form haben:

```
SELECT * FROM report_function(from, to);
```

Lösung



```

1  CREATE FUNCTION report_function(date_from date, date_to date)
2  RETURNS TABLE (
3      category_name text,
4      mon int,
5      tue int,
6      wed int,
7      thu int,
8      fri int,
9      sat int,
10     sun int,
11     total int
12 )
13 AS $$
14     SELECT    category.name AS category_name,
15              COUNT(*) FILTER (WHERE
16                  EXTRACT(ISODOW FROM rental.rental_date) = 1) AS mon,
17              COUNT(*) FILTER (WHERE
18                  EXTRACT(ISODOW FROM rental.rental_date) = 2) AS tue,
19              COUNT(*) FILTER (WHERE
20                  EXTRACT(ISODOW FROM rental.rental_date) = 3) AS wed,
21              COUNT(*) FILTER (WHERE
```

²<https://www.postgresql.org/docs/15/sql-createfunction.html>


```

22             EXTRACT(ISODOW FROM rental.rental_date) = 4) AS thu,
23     COUNT(*) FILTER (WHERE
24             EXTRACT(ISODOW FROM rental.rental_date) = 5) AS fri,
25     COUNT(*) FILTER (WHERE
26             EXTRACT(ISODOW FROM rental.rental_date) = 6) AS sat,
27     COUNT(*) FILTER (WHERE
28             EXTRACT(ISODOW FROM rental.rental_date) = 7) AS sun,
29     COUNT(*) AS total
30 FROM     rental
31 INNER JOIN inventory
32 ON        rental.inventory_id = inventory.inventory_id
33 INNER JOIN film_category
34 ON        inventory.film_id = film_category.film_id
35 INNER JOIN category
36 ON        film_category.category_id = category.category_id
37 WHERE     rental.rental_date::date BETWEEN date_from AND date_to
38 GROUP BY  category.name
39 ORDER BY  category.name
40 $$ LANGUAGE SQL;

```

Abgabe



exercise3.py oder

exercise3.c oder

Exercise3.java

Wenn Sie Ihr Programm über mehrere Source-Dateien verteilt haben, geben Sie alle davon mit ab.

3c.sql: Code der tabellenwertigen Funktion

Lösung



Siehe exercise3.c, exercise3.py, Exercise3.java.

Hinweis



Für die Lösung dieser Aufgabe verwenden Sie low-level Schnittstellen, um über einen Cursor auf die Datenbank zuzugreifen. In der Praxis verwendet man heute Werkzeuge wie Objekt-Relational-Mapper (ORM), um sauberer auf die Datenbank zuzugreifen. Sie werden in späteren Lehrveranstaltungen im Studium lernen, wie man diese verwendet.

Wichtig: Laden Sie bitte Ihre Lösung in OLAT hoch und geben Sie mittels der Ankreuzliste auch unbedingt an, welche Aufgaben Sie gelöst haben. Die Deadline dafür läuft am Vortag des Proseminars um 16:00 ab.