database
research group

Datenbanken II

Wintersemester 2019/20

Praktische Aufgabe 4

Der Inhalt dieser Übung wird im *3. Quiz* überprüft.

---

1 Anfrageoptimierung in *PostgreSQL*

In dieser praktischen Übung betrachten wir Methoden, die es *PostgreSQL* ermöglichen, Anfragen effizient zu beantworten. Im Speziellen geht es dabei um den sogenannten *Optimizer*. Um einen Überblick der gesamten Architektur zu bekommen, betrachten wir zuerst die einzelnen Komponenten von *PostgreSQL*.

1.1 Komponenten von *PostgreSQL*

Zwischen dem Absetzen einer Anfrage und der Rückgabe des Ergebnisses durchlaufen die Anfrage und die Daten mehrere Komponenten. Die folgende Aufzählung beschreibt die wichtigsten Komponenten und die Reihenfolge, in der die Anfrage bearbeitet wird:

1. **Empfangen der Anfrage:** Zu Beginn muss ein *Client-Prozess* eine Verbindung zum gewünschten *PostgreSQL* Server herstellen. Dies ist grundsätzlich mit allen Programmen möglich, die das *PostgreSQL* Kommunikations-Protokoll implementiert haben. Für viele der gängigen Programmiersprachen gibt es Bibliotheken, die die Kommunikation mit dem *PostgreSQL* Server erlauben, zum Beispiel `libpq` für *C* oder den `JDBC driver` für *Java*. Anschließend kann die Anfrage als Text zum Server übertragen und im nächsten Schritt weiterverarbeitet werden.
2. **Parser stage:** Die *parser stage* besteht aus zwei Teilen mit seperaten Aufgaben. Zuerst validiert der *Parser*, ob die gegebene Anfrage syntaktisch korrekt ist. Ist dies der Fall, wird aus der Anfrage ein entsprechender *parse tree* erzeugt. Der Parser holt keine zusätzlichen Informationen aus dem System Katalog, somit fehlt dem parse tree jegliche Semantik. Diese semantische Interpretation erfolgt nun

im zweiten Schritt, dem *transformation process*. Hier wird nun für jeden Knoten im übergebenen parse tree analysiert, um welche Tabelle oder um welchen Operator es sich handelt oder welchen Typ eine gewisse Spalte hat. Der daraus resultierende Baum wird als *query tree* bezeichnet.

3. **Optimizer:** Der *Optimizer* nimmt nun den erzeugten query tree und versucht ihn durch Umformung effizienter zu machen, ohne das Ergebnis zu verändern. Der query tree mit den geringsten Kosten wird anschließend als *execution plan* zur Ausführung weitergegeben. Auf die Details der Techniken und Vorgänge in *PostgreSQL* wird in den nächsten Kapiteln genauer eingegangen.
4. **Executor:** Der *executor* führt rekursiv die Schritte des übergebenen *execution plans* aus und erhält somit Zeile für Zeile das Ergebnis der Anfrage. Im Zuge dessen werden eventuell Daten von der Festplatte gelesen, sortiert, vereinigt oder Bedingungen geprüft.

1.2 *PostgreSQL* Optimizer

Das Ziel des Optimizers ist es, den query tree zu finden, der zur effizientesten Ausführung der Anfrage führt. Abhängig von der Anfrage kann es eine hohe Anzahl an verschiedenen query trees geben die zum gleichen Ergebnis führen. Solange die Kosten aller möglichen query trees in plausibler Zeit berechnet werden können, werden diese berechnet und der effizienteste Ansatz ausgewählt.

Sollte die Kostenberechnung aller query trees nicht mehr möglich sein, wird in *PostgreSQL* ein spezieller *genetic query optimizer* verwendet, der auf genetischen Algorithmen basiert. Dieser Ansatz ist jedoch nicht im Fokus dieser praktischen Übung.

1.2.1 Erstellen von möglichen Execution Plans

Lesen der Daten: Beim Erstellen eines execution plans muss zu Beginn festgelegt werden, wie die Tabelle selbst gelesen wird. Eine Methode, die immer funktioniert, ist das sequentielle Lesen der Relation. Vorhandene Indizes müssen jedoch beachtet werden, da dadurch der Leseprozess effizienter werden kann.

Joinen von Tabellen: Im Bezug auf Optimierung ist der Join einer der problematischten Operatoren. Grund dafür ist, dass die Anzahl an query trees exponentiell mit der Anzahl an Joins wächst (Join-Reihenfolge). Im Falle, dass die Anzahl an Joins einen festgelegten Grenzwert überschreitet, wird der *genetic query optimizer* verwendet. Ansonsten werden, wie üblich, alle möglichen query trees berechnet. Zusätzlich zur Reihenfolge kann in *PostgreSQL* einer der folgenden Join-Algorithmen verwendet werden:

- Nested Loop Join
- Merge Join
- Hash Join

Der berechnete query tree mit den geringsten Kosten wird als execution plan an den executor übergeben. Dieser Plan besteht aus sequentiellen oder Index Scans, nested loop, merge oder hash Joins und zusätzlichen Informationen wie Sortieralgorithmen oder Aggregatfunktionen (Durchschnitt, Maximum, etc.).

2 Analyse von Execution Plans

In diesem Teil werden wir die execution plans von bestimmten Anfragen an die *IMDB* Datenbank analysieren, die von *PostgreSQL* berechnet wurden. Zusätzlich werden wir Indizes erstellen um die Anfragen zu optimieren. Dabei können wir sehen, wie der Optimizer die vorhanden Informationen nutzen kann, um den execution plan zu verbessern. Um dies Durchführen zu können, benötigen wir folgende Befehle:

- `EXPLAIN`: Zeigt den Ausführungsplan an *ohne* die nachfolgende Anfrage auszuführen.

  Beispiel:
  ```
  EXPLAIN SELECT * FROM names WHERE birthyear > 1900;
  ```

- `EXPLAIN ANALYZE`: Zeigt den Ausführungsplan an und führt die nachfolgende Anfrage aus.

  Beispiel:
  ```
  EXPLAIN ANALYZE SELECT * FROM names WHERE deathyear > 1900;
  ```

- `CREATE INDEX`: Erstellt einen Index auf einem definierten Attribut einer definierten Tabelle. Es kann auch die Datenstruktur, die für den Index verwendet wird, vorgegeben werden. Wenn ein Index auf einer Tabelle erstellt wird, hat dies keinerlei Auswirkungen auf die Sortierung der betroffenen Tabelle.

  Beispiel:
  ```
  CREATE INDEX names_deathyear_idx ON names(deathyear);
  ```

- `CLUSTER`: Um einen Clustering Index auf einer Tabelle zu erstellen (d.h. die Tabelle physisch zu sortieren), muss zuerst ein Index auf dem entsprechenden Attribut erstellt werden. Danach kann die Tabelle nach diesem Index geclustered werden.

  Beispiel:
  ```
  CREATE INDEX names_deathyear_idx ON names(deathyear);
  CLUSTER names USING names_deathyear_idx;
  ```

- `ANALYZE`: Aktualisiert die Statistiken einer definierten Tabelle. Die ist vor allem nach `CLUSTER` wichtig, da die Datenbank sonst nicht weiß, dass eine Umsortierung der betroffenen Tabelle stattgefunden hat.

  Beispiel:
  ```
  ANALYZE names;
  ```

2.1 Punktanfragen

Zuerst betrachten wir folgende Punktanfrage auf die Tabelle `titles` in Beispiel 1.

Beispiel 1 *Execution plan der gegebenen Punktanfrage.*

```
EXPLAIN ANALYZE SELECT * FROM titles
  WHERE runtimeMinutes = 1000;

                        QUERY PLAN
-----------------------------------------------------------
Gather  (cost=1000.00..130579.50 rows=2067 width=112)
        (actual time=6.412..5436.996 rows=2396 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on titles
        (cost=0.00..129372.80 rows=861 width=112)
        (actual time=7.200..5425.023 rows=799 loops=3)
       Filter: (runtimeminutes = 1000)
       Rows Removed by Filter: 1815159
Planning time: 0.224 ms
Execution time: 5437.538 ms
```

Ohne Index oder Ordnung der Daten kann diese Anfrage nur mit einem sequentiellen Scan der gesamten Tabelle beantwortet werden. Um Punktanfragen zu optimieren, eignet sich ein Hash-Index, welchen wir nun mit folgendem Befehl erstellen:

```
CREATE INDEX titles_runtimeminutes_idx
  ON titles USING hash(runtimeMinutes);
```

Wie in Beispiel 2 dargestellt, wird nun aufgrund des Indizes ein *Bitmap Index Scan* verwendet. Dabei werden mittels des Indizes die Seiten aller Tupel markiert, die in der Ergebnismenge enthalten sind. Diese Liste von markierten Seiten werden an den *Bitmap Heap Scan* übergeben, anschließend gelesen und das Ergebnis zurückgegeben. Die Effizienz dieser Optimierung kann von der Ausführungszeit (execution time) abgelesen werden.

Beispiel 2 *Execution plan der optimierten Punktanfrage.*

```
EXPLAIN ANALYZE SELECT * FROM titles
  WHERE runtimeMinutes = 1000;

                        QUERY PLAN
-----------------------------------------------------------
Bitmap Heap Scan on titles
  (cost=56.01..7392.19 rows=2066 width=112)
  (actual time=2.260..19.561 rows=2396 loops=1)
       Recheck Cond: (runtimeminutes = 1000)
       Heap Blocks: exact=2071
       ->  Bitmap Index Scan on titles_runtimeminutes_idx
        (cost=0.00..55.49 rows=2066 width=0)
        (actual time=1.500..1.500 rows=2396 loops=1)
              Index Cond: (runtimeminutes = 1000)
     Planning time: 0.304 ms
     Execution time: 19.906 ms
```

2.2 Bereichsanfragen

Anstatt einer Punktanfrage betrachten wir nun die Bereichsanfrage in Beispiel 3 an die Tabelle names.

Beispiel 3 *Execution plan der gegebenen Bereichsanfrage.*

```
EXPLAIN ANALYZE SELECT * FROM names
  WHERE birthyear > 1968 AND birthyear < 1970;
                        QUERY PLAN
-----------------------------------------------------------
Gather
  (cost=1000.00..217535.63 rows=5218 width=119)
  (actual time=2.386..1104.270 rows=6409 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       ->  Parallel Seq Scan on names
        (cost=0.00..216013.83 rows=2174 width=119)
        (actual time=0.836..1089.303 rows=2136 loops=3)
           Filter: ((birthyear > 1968) AND (birthyear < 1970))
           Rows Removed by Filter: 2994868
     Planning time: 0.141 ms
     Execution time: 1105.068 ms
```

Wie auch zuvor bei der Punktanfrage, muss die gesamte Tabelle gelesen werden, da weder ein Index noch eine Sortierung vorhanden sind. Da für Bereichsanfragen ein B+ Baum Index viel effizienter ist als ein Hash-Index, erstellen wir folgenden Index auf das Attribut birthyear.

```
CREATE INDEX names_birthyear_idx
  ON names USING btree(birthyear);
```

Dadurch ändert sich die Strategie im execution plan zu einem *Index Scan*, siehe Beispiel 4. Wir können nun den Index verwenden, um das erste Tupel zu finden, welches birthyear > 1968 erfüllt. Anschließend werden im Index solange die Blattknoten gescannt, bis birthyear > 1970 ist. Die jeweiligen Datenblöcke müssen jedoch gelesen werden, da wir alle Informationen eines Tupels benötigen.

Beispiel 4 *Execution plan der gegebenen Bereichsanfrage.*

```
EXPLAIN ANALYZE SELECT * FROM names
  WHERE birthyear > 1968 AND birthyear < 1970;
                        QUERY PLAN
-----------------------------------------------------------
Index Scan using names_birthyear_idx on names
  (cost=0.43..15493.81 rows=5218 width=119)
  (actual time=1.299..10.893 rows=6409 loops=1)
    Index Cond: ((birthyear > 1968) AND (birthyear < 1970))
Planning time: 2.218 ms
Execution time: 11.515 ms
```

3 Zusätzliche Informationen

Das Kapitel zur Dateiorganisation in *PostgreSQL* basiert auf den Inhalt der folgenden Websites:

- PostgreSQL Dokumentation Overview:
  `https://www.postgresql.org/docs/current/overview.html`
- PostgreSQL Dokumentation `EXPLAIN`:
  `https://www.postgresql.org/docs/current/sql-explain.html`
- PostgreSQL Dokumentation using `EXPLAIN`:
  `https://www.postgresql.org/docs/current/using-explain.html`
- PostgreSQL Dokumentation `CREATE INDEX`:
  `https://www.postgresql.org/docs/current/sql-createindex.html`
- PostgreSQL Dokumentation `BTREE`:
  `https://www.postgresql.org/docs/current/pageinspect.html#id-1.11.7.31.6`
- PostgreSQL Dokumentation `CLUSTER`:
  `https://www.postgresql.org/docs/current/sql-cluster.html`