database
research group

Datenbanken II

Wintersemester 2019/20

Praktische Aufgabe 3

Der Inhalt dieser übung wird im *2. Quiz* überprüft.

---

1 Indexstrukturen in *PostgreSQL*

1.1 Buffers

In den Beispielen der Übungsblätter berechnen wir die Anzahl an Blocken, die zu die gelesen werden müssen, um eine gegebene Anfrage zu beantworten. In einem realen System werden dabei die Blöcke zuerst von der Festplatte und dann in den Arbeitsspeicher geladen. Im nächsten Schritt betrachten wir, wie man die Anzahl an gelesenen Blöcken in *PostgreSQL* auslesen kann.

Betrachten wir dazu die Anfrage an die *IMDB* Datenbank in Beispiel 1.

Beispiel 1 *Anfrage zum Auslesen aller Vornamen von Personen die nach 1940 geboren wurden und vor 1945 gestorben sind.*

```
SELECT primaryname FROM names
  WHERE birthyear > 1940 AND deathyear < 1945;
```

Die erste Ausfürung dieser Anfrage kann etwas Zeit in Anspruch nehmen. Füren wir die Anrage erneut aus, können wir feststellen, dass sich die Ausfürungzeit wesentlich verkürzt. Der Grund dafür ist, dass zuerst die Daten von der Festplatte in den Arbeitsspeicher geladen werden müssen. Bei der zweiten Ausfürung sind die Daten sehr wahrscheinlich noch im Speicher. Um dies zu überprüfen, können wir uns die Anzahl an Seiten, die von der Festplatte gelesen werden müssen, auslesen.

Um dies korrekt durchführen zu können, müssen wir zuerst den *PostgreSQL*-Server terminieren, den Betriebssystem-Cache löschen und den *PostgreSQL*-Server erneut starten. In Beispiel 2 finden Sie Kommandozeilenbefehle die dies in Debian-basierten Systemen durchführen.

Beispiel 2 *Befehle zum Betriebssystem-Cache löschen und* PostgreSQL*-Server neustarten.*

```
sudo service postgresql stop free && sync && \
echo 3 > sudo /proc/sys/vm/drop_caches && free && \
sudo service postgresql start
```

Führen Sie die folgende, leicht modifizierte Anfrage aus Beispiel 3 aus. Mit `EXPLAIN` wird der Ausführungsplan einer Anfrage ausgegeben, `ANALYZE` gibt Statistiken und die Ausführungszeit einer Anfrage ausgegeben und `BUFFERS` erweitert diese Informationen mit Statistiken über die Anzahl an Blöcken, die von der Festplatte in den Arbeitsspeicher gelesen wurden.

Beispiel 3 *Informationen zur Ausführung einer Anfrage mit* `EXPLAIN (ANALYZE, BUFFERS)`.

```
EXPLAIN (ANALYZE,BUFFERS) SELECT primaryname FROM names
  WHERE birthyear > 1940 AND deathyear < 1945;

                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=1000.00..232015.16 rows=341 width=14)
         (actual time=51.611..548.566 rows=2 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared read=170645
   ->  Parallel Seq Scan on names
       (cost=0.00..230981.06 rows=142 width=14)
       (actual time=202.486..536.846 rows=1 loops=3)
         Filter: ((birthyear>1940) AND (deathyear<1945))
         Rows Removed by Filter: 3217528
         Buffers: shared read=170645
 Planning Time: 3.818 ms
 Execution Time: 548.695 ms
```

In der Ausgabe von Beispiel 3 (`Buffers: shared read=170645`), sehen Sie das 170645 Blöcke von der Festplatte gelesen wurden.

Nachdem die Anfrage erneut ausgeführt wurde, ändert sich die Ausgabe wie folgt: `Buffers: shared hit=96 read=170549`. Dies bedeutet, dass 96 Blöcke bereits im Buffer gefunden wurden.

Der `EXPLAIN` Befehl zeigt uns zusätzlich den *Query Plan*. Für unsere Anfrage ist dies ein `-> Parallel Seq Scan on names`. Es gibt keine bessere Strategie als ein sequentielles Durchsuchen aller Daten, da kein Index auf `birthyear` oder `deathyear` der Tabelle `names` existiert. Dies kann mit dem Befehl `\d names` verifiziert werden. In der Ausgabe sehen wir im Bereich für Indizes lediglich `"names_pkey"PRIMARY KEY, btree (nconst)`. Dies bedeutet, dass ein `btree` Index auf dem Attribut `nconst` existiert, was jedoch nicht zur effizienten Beantwortung unserer Anfrage verwendet werden kann.

1.2 Indexerzeugung

Der Befehl in Beispiel 4 kann verwendet werden, um einen `btree` Index auf dem Attribut `birthyear` zu erstellen. Neben dem B-tree existieren in *PostgreSQL* die Indextypen `Hash`, `GiST`, `SP-GiST`, `GIN` und `BRIN`. Details dazu finden Sie in der *PostgreSQL* Dokumentation.

Beispiel 4 *Anlegen eines* `btree` *Index auf* `birthyear` *in der Tabelle* `names`.

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

In der Ausgabe des Befehls `\d names` sehen wir nun zusätzlich den Indexeintrag `"names_birthyear_idx"btree (birthyear)`.

Stellen wir nun erneut die Anfrage aus Beispiel 1, kann der Index auf `birthyear` verwendet werden und wir bekommen eine Ausgabe wie in Beispiel 5. Um zu vermeiden, dass sich Blöcke bereits im Cache befinden, löschen Sie den Cache wie in Beispiel 2 beschrieben.

Beispiel 5 *Informationen zur Beantwortung der Anfrage aus Beispiel 1 unter Verwendung eines Indizes.*

```
EXPLAIN (ANALYZE, BUFFERS) SELECT primaryname FROM names
  WHERE birthyear > 1940 AND deathyear < 1945;

                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=6690.39..229744.14 rows=341 width=14)
         (actual time=125.949..498.334 rows=2 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared read=109948 written=1
   ->  Parallel Bitmap Heap Scan on names
       (cost=5690.39..228710.04 rows=142 width=14)
       (actual time=229.595..479.904 rows=1 loops=3)
         Recheck Cond: (birthyear > 1940)
         Rows Removed by Index Recheck: 1111624
         Filter: (deathyear < 1945)
         Rows Removed by Filter: 107729
         Heap Blocks: exact=12277 lossy=19779
         ->  Bitmap Index Scan on names_birthyear_idx
             (cost=0.00..5690.30 rows=307982 width=0)
             (actual time=78..78 rows=323188 loops=1)
               Index Cond: (birthyear > 1940)
               Buffers: shared read=886
 Planning Time: 0.241 ms
 Execution Time: 498.388 ms
```

Nun verwenden wir unseren Index `names_birthyear_idx` und markieren alle Seiten mit Tupeln, die die Anfrage ($birthyear > 1940$) erfüllen. Anschließend werden alle diese markierten Seiten gelesen und Bedingung ($deathyear < 1945$) wird überprüft.

Diese Analyse kann für beliebige Anfragen durchgeführt werden. Versuchen Sie die Ausfürungspläne anderer Anfragen zu interpretieren, zum Beispiel: Geben Sie alle Personen aus, die bei dem Film *The Lion King* mitgewirkt haben.

Beispiel 6 *Befehl zum Auslesen aller Personen, die bei dem Film* The Lion King *mitgewirkt haben.*

```
SELECT primaryname FROM names
  NATURAL JOIN principals NATURAL JOIN titles
  WHERE primarytitle = 'The Lion King'
    AND startyear=1994 AND titletype='movie';
```

Weiters können Sie sich überlegen, ob ein Index erzeugt werden kann, der die Ausfürung der Anfrage beschleunigen könnte.

1.3 B-trees in *PostgreSQL*

Wir betrachten die Indexstruktur *B-tree* und ihre Implementierung in *PostgreSQL* basierend auf dem von uns erstellten Index `names_birthyear_idx`. Wir können die Funktion `bt_metap` verwenden um Details zu diesem Index auszugeben.

Beispiel 7 *Informationen des erstellten* `btree` *Index* `names_birthyear_idx`.

```
SELECT * FROM bt_metap('names_birthyear_idx');

 magic  | version | root | level | fastroot | fastlevel
--------+---------+------+-------+----------+-----------
 340322 |       3 |  290 |     2 |      290 | 2
```

In Beispiel 7 sehen wir, neben weiteren Informationen, dass der Baum des Indizes `names_birthyear_idx` 2 Levels hat und die ID der Wurzel 290 ist. Die Anzahl an Level ist unerwartet niedrig, wenn man bedenkt, dass `names` 4 Millionen Tupel hat.

Im nächsten Schritt sehen wir uns die Wurzel des B-Trees genauer an.

Beispiel 8 *B-Tree details.*

```
SELECT * FROM bt_page_items('names_birthyear_idx', 290);

 itemoff |    ctid     | itemlen |           data
---------+-------------+---------+-------------------------
       1 | (3,0)       |       8 |
       2 | (289,24)    |      16 | 85 07 00 00 00 00 00 00
       3 | (575,7)     |      16 | 9d 07 00 00 00 00 00 00
       4 | (860,31)    |      16 | b1 07 00 00 00 00 00 00
       5 | (1145,49)   |      16 | bf 07 00 00 00 00 00 00
       6 | (1430,2)    |      16 |
       7 | (1715,28)   |      16 |
       8 | (2000,19)   |      16 |
       9 | (2285,21)   |      16 |
      10 | (2570,45)   |      16 |
     ... |     ...     |     ... |
```

Das Feld `data` gibt den kleinsten Wert an, der in dem Teilbaum der Seite `ctid` vorkommt. Zum Beispiel können wir aus der zweiten Zeile entnehmen, dass der kleinste Wert, im Teilbaum der Seite 289 vorkommt, 85 ist. Dies entspricht dem Geburtsjahr 1925.

Mit der Funktion `bt_page_items` können wir den Baum schrittweise nach unten verfolgen und die Indexseiten analysieren. Der Befehl `bt_page_stats` zeigt das Attribut `type`, welches anzeigt, ob eine Seite eine *root page* `r`, *internal page* `i` oder *leaf page* `l` ist. Der Befehl kann wie folgt verwendet werden:

`SELECT * FROM bt_page_stats('names_birthyear_idx', 3);`.

Root und internal pages speichern in `ctid` Indexseiten und leaf pages speichern in `ctid` Datenseiten.

Als nächstes versuchen wir, das Tupel mit dem niedrigsten Geburtsjahr (`birthyear`) zu finden. Dies unterscheidet sich anhängig von der Datensatzversion.

Die root page von `names_birthyear_idx` ist 290. Um nun die erste internal page der Wurzel zu finden kann folgender Befehl verwendet werden:

`SELECT ctid FROM bt_page_items('names_birthyear_idx', 290) LIMIT 1;`.

Das Ergebnis ist `(3,0)`. Weiters können wir nun die erste leaf page der Seite 3 mit folgendem Befehl finden: `SELECT * FROM bt_page_items('names_birthyear_idx', 3);`. Der erste Eintrag zeigt auf die erste Seite der Geschwisterseite.

In unserem Fall ist dies die Seite mit `ctid=(1,0)`. Der zweite Eintrag auf dieser leaf page bringt uns zu dem Tupel mit dem kleinsten Geburtsjahr (`birthyear`), Dies können wir mit dem Befehl `SELECT * FROM bt_page_items('names_birthyear_idx', 1);` auslesen. Das Tupel mit dem niedrigsten Geburtsjahr hat `ctid=(3918,32)`. Das bedeutet, es ist der 32. Eintrag auf der Seite 3918 der Tabelle `names`. Dies können wir mit `SELECT ctid, * FROM names where ctid = '(3918,32)';` abfragen.

1.4 Sortieren der Daten

In *PostgreSQL* ist es möglich, die Daten mit dem CLUSTER Befehl physisch zu sortieren. Dafür ist es nötig, einen Index auf das zu sortierende Attribut zu erstellen. In unserem Falle verwenden wir den Index names_birthyear_idx, um die Daten der Tabelle names nach dem Geburtsjahr (birthyear) zu sortieren. Der nötige Befehl ist in Beispiel 9 zu sehen.

Beispiel 9 *Befehl zum Sortieren der Tabelle* names *nach dem Attribut* birthyear. names.

```
CLUSTER names USING names_birthyear_idx;
```

Um die Daten zu sortieren, ließt *PostgreSQL* all Tupel und schreibt sie geordnet auf die Festplatte zurück. Für große Tabellen kann dies einige Zeit in Anspruch nehmen. Nun sehen Sie in der Ausgabe des Befehls \d names den Zusatz CLUSTER bei unserem Index names_birthyear_idx.

Nun können wir vergleichen, wie sich die Sortierung auf die Auswertung unserer Beispielanfrage (siehe Beispiel 10) auswirkt. Um zu vermeiden, dass sich Blöcke bereits im Cache befinden, löschen Sie den Cache wie in Beispiel 2 beschrieben.

Beispiel 10 *Informationen zur Ausführung einer Anfrage mit sortierten Daten (*birthyear*).*

```
EXPLAIN (ANALYZE, BUFFERS) SELECT primaryname FROM names
  WHERE birthyear > 1940 AND deathyear < 1945;

                        QUERY PLAN
-----------------------------------------------------------
 Gather  (cost=6690.39..230212.66 rows=341 width=14)
         (actual time=416.478..419.296 rows=2 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared read=8194
   ->  Parallel Bitmap Heap Scan on names
       (cost=5690.39..229178.56 rows=142 width=14)
       (actual time=282.489..413.367 rows=1 loops=3)
         Recheck Cond: (birthyear > 1940)
         Filter: (deathyear < 1945)
         Rows Removed by Filter: 107729
         Heap Blocks: exact=2449
         Buffers: shared read=8194
         ->  Bitmap Index Scan on names_birthyear_idx
             (cost=0.00..5690.30 rows=307982 width=0)
             (actual time=17..17 rows=323188 loops=1)
               Index Cond: (birthyear > 1940)
               Buffers: shared read=886
 Planning Time: 6.412 ms
 Execution Time: 419.972 ms
```

Mit Sortierung werden lediglich 8194 Blöcke gelesen, im Vergleich zu 109948 Blöcken in Beispiel 5. Auch wenn die Strategie zur Bearbeitung unverändert bleibt, sind nun alle Tupel mit gleichen Geburtsjahr nacheinander abgespeichert. Ohne Sortierung waren die Tupel auf vielen unterscheidlichen Seiten abgespeichert.

2 Zusätzliche Informationen

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

- 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`