DB JPQL Queries

Hammann, Wilpert

July 2020

1 Erklärung zu den Queries

Innerhalb dieses Protokolls zeigen wir unsere Lösung für die Vorbereitung des 5. Praktikums auf und erläutern die entstandenen Explains

1. Ausgabe aller Spieler (Spielername), die in einem bestimmten Zeitraum gespielt hatten

   Unsere erstellte JPQL-Query sieht wie folgt aus:

   ```
   SELECT g.player.name
   FROM Game g
   WHERE g.starttime BETWEEN :start AND :end
   GROUP BY g.player.name
   ```

   Die Parameter start und end werden dann mit dem ausgewählten Datum gesetzt, das Ergebnis ist eine Liste mit den Spielernamen in Strings

   Zuerst wird die Game Tabelle mit dem Datum gefiltert und die Player Tabelle gehasht. Und anschließend mit einem Hash Inner join verknüpft danach wird sie sortiert damit sie gegrouped werden kann.

   Den entsprechende Explain kann man Abbildung 1 entnehmen

   Optimierung:
   Um die erste Query zu optimieren haben wir zwei zusätzliche Indexe erstellt. Einmal einen für den playername in der Game Tabelle und einen weiteren für den Starttermin in der Tabelle Game. Dadurch sind die Kosten von 17367 auf 10870 gesunken. (Explain siehe Abbildung 5)

2. Ausgabe zu einem bestimmten Spieler: Alle Spiele(ID,Datum), sowie die Anzahl der korrekten Antworten pro Spiel mit Angabe der Gesamtanzahl der Fragen pro Spiel bzw. alternativ den Prozentsatz der korrekt beantworteten Fragen

   Unsere erstellte JPQL-Query für die zweite Analyseaufgabe sieht wie folgt aus:

```
SELECT g.id, g.starttime, g.endtime, COUNT(gq), SUM(gq.givenAnswer)
FROM Game g, GameQuestion  gq
WHERE gq.game = g AND g.player.name= :playerName
GROUP BY g.id
```

Der Parameter playerName wird mit dem gewünschten eingegebenen Namen gesetzt, das Ergebnis dieser Query ist eine List von Object arrays : List<Object[]>

Anfangs wird die Game Tabelle gefiltert mit dem gesuchten Spielernamen ,diese wird dann gehasht um mit einem Hash Inner Join mit der Game Tabelle verbunden zu werden. Anschließend wird noch aggregiert um COUNT(gq) und SUM(gp.givenAnswer) zusammenzurechnen.

In Abbildung 2 kann man den Explain sehen

Optimierung:
Um die zweite Query zu optimieren haben wir einen zusätzlichen Indexe erstellt, nämlich für die gameid in der Tabelle Gamequestion. Dadurch sind die Kosten von 92878 auf 1117 gesunken. (Explain siehe Abbildung 6)

3. Ausgabe aller Spieler mit Anzahl der gespielten Spiele, nach Anzahl absteigend geordnet

   Unsere erstellte JPQL-Query für die dritte Analyseaufgabe sieht wie folgt aus:

   ```
   SELECT g.player.name, COUNT(g)
   FROM Game g
   GROUP BY g.player
   ORDER BY COUNT(g)
   DESC
   ```

   Das Ergebnis dieser Query ist wieder eine List von Object arrays : List<Object[]>

   Zuerst wird die Player Tabelle gehasht um sie mit einem Hash Inner Join mit den Games zu joinen anschließend wird durch die Aggregatsfunktion das Group By realisiert welches Einträge mit dem selben Spieler zusammenfasst und dessen Spiele zusammenrechnet. Anschließend wird das Ergebnis noch absteigend sortiert.

   In Abbildung 3 wird der Explain gezeigt

   Optimierung:
   Um die dritte Query zu optimieren haben wir einen zusätzlichen Indexe erstellt, nämlich für den playername in der Tabelle Game. Die Kosten sind dadurch nicht weniger geworden. (Explain siehe Abbildung 7)

4. Ausgabe der am meisten gefragten Kategorie, oder alternativ, die Beliebtheit der Kategorien nach Anzahl der Auswahl absteigend sortiert

Unsere erstellte JPQL-Query für die vierte Analyseaufgabe sieht wie folgt aus:

```
SELECT c.name,
COUNT(c)
FROM Category c, Question q, GameQuestion gq
WHERE c=q.category AND q=gq.question
GROUP BY c.name
ORDER BY COUNT(c)
DESC
```

Das Ergebnis dieser Query ist wieder eine List von Object arrays : List<Object[]>
Zuerst werden alle 3 Tabellen Sequenziell gescant, um dann verhasht zu werden. Die gamequestion und question Tabelle werden miteinander in einem Hash Inner Join verbunden. Danach wird das selbe nochmal mit der category Tabelle gemacht. Die Aggregat Funktion kommt durch das benutzte GROUP By,ORDER BY und COUNT(c). Zum Schluss findet eine Sortierung statt um die absteigende Reihenfolge einzuhalten.
Das Explain dazu ist in Abbildung 4 zu sehen

Optimierung:
Um die vierte Query zu optimieren haben wir drei zusätzliche Indexe erstellt, einmal für den Namen der Category und dann für die Fremdschlüssel der Join Bedingungen . Die Kosten sind dadurch nicht weniger geworden. (Explain siehe Abbildung 8)

Optimierungen:

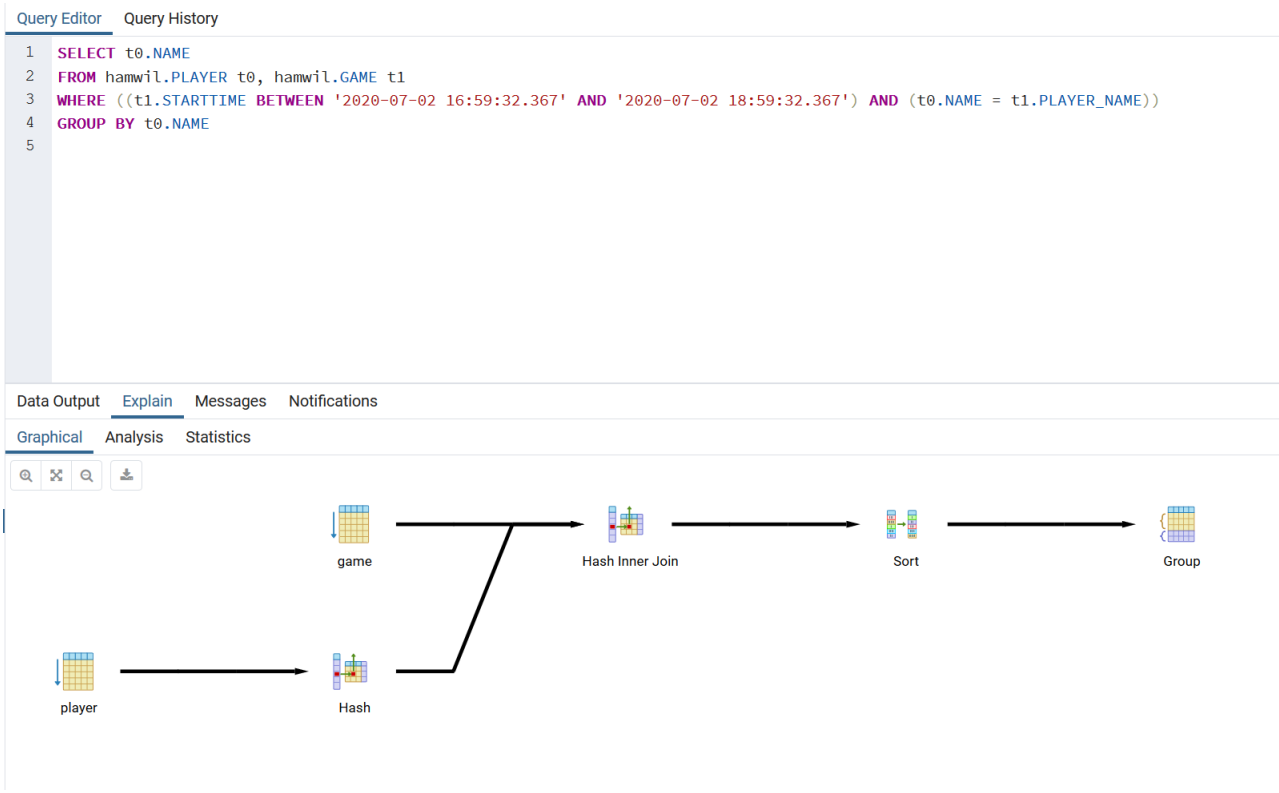

Abbildung 1: Explain zu der ersten Query

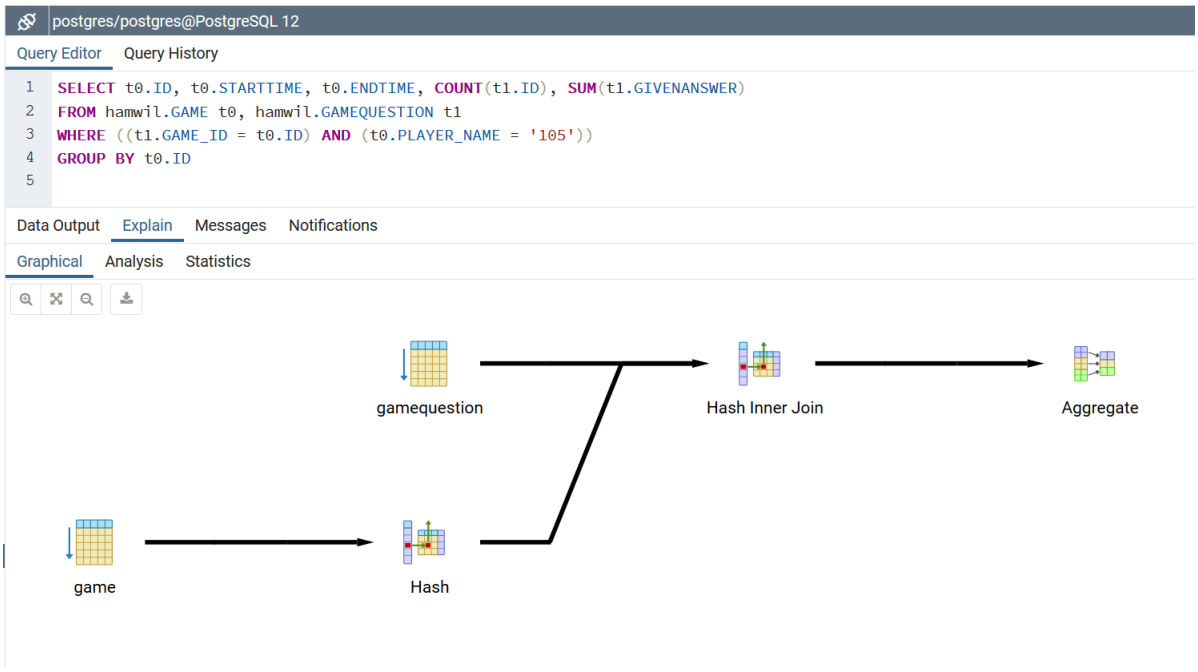

Abbildung 2: Explain zu der zweiten Query

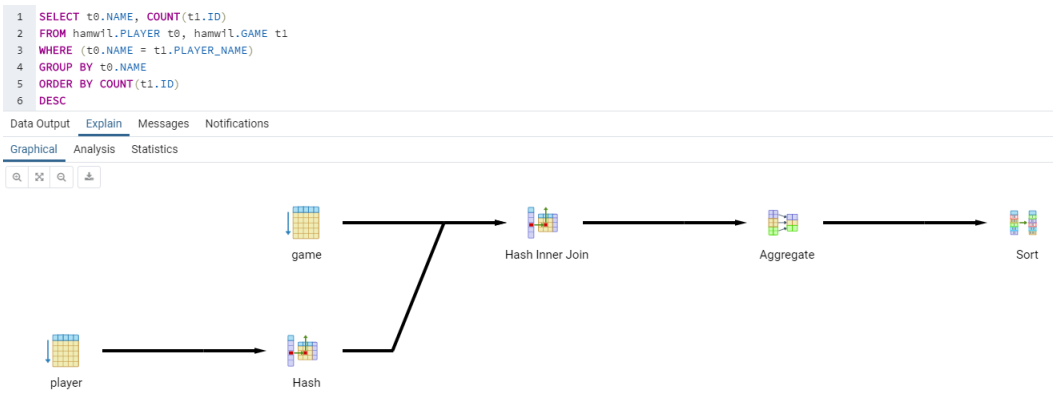

Abbildung 3: Explain zu der dritten Query

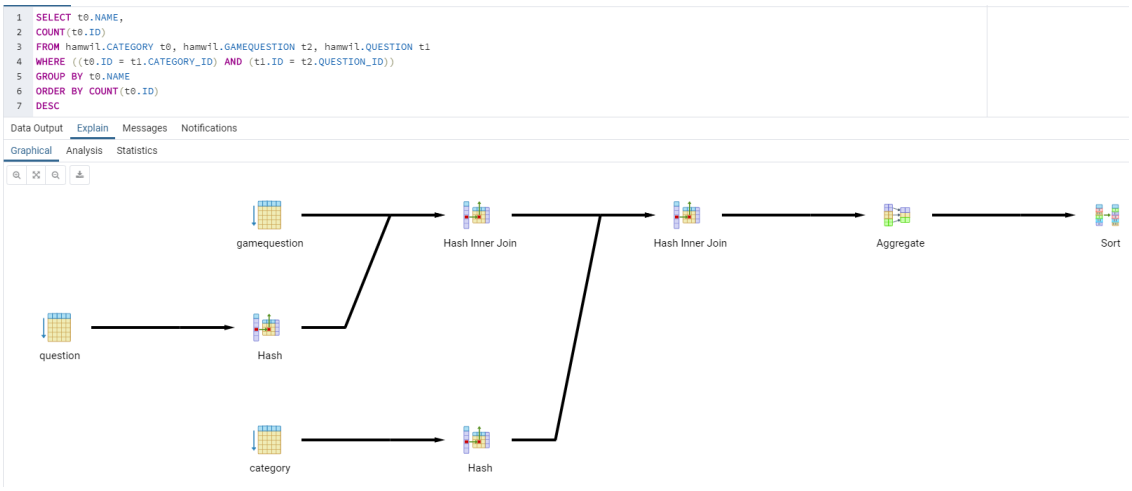

Abbildung 4: Explain zu der vierten Query

Abbildung 5: Optimierung der ersten Query

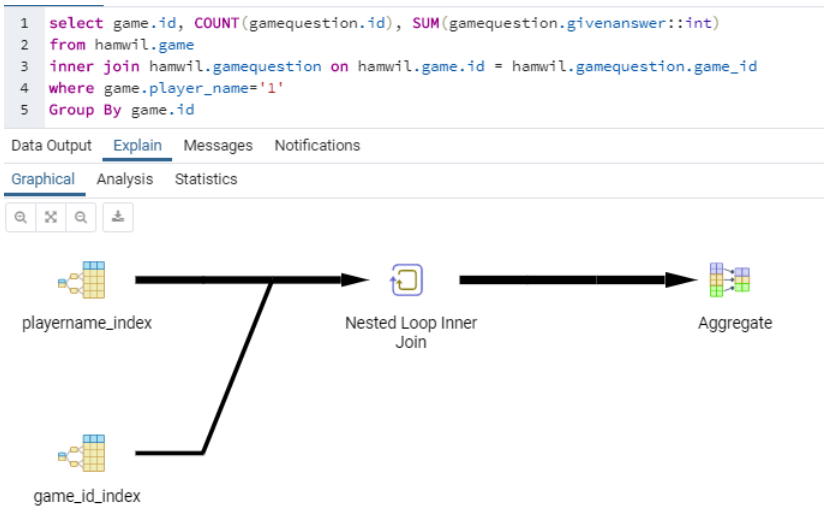

Abbildung 6: Optimierung der zweiten Query

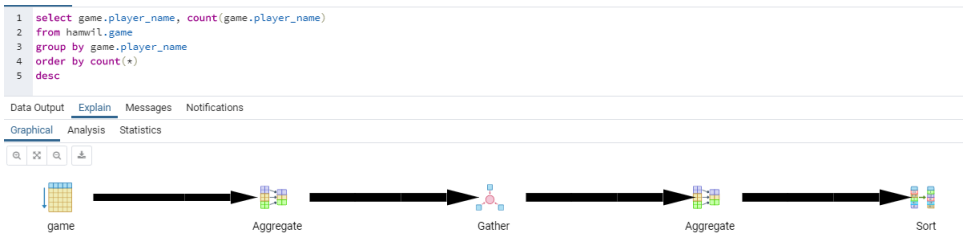

Abbildung 7: Optimierung der dritten Query

Abbildung 8: Optimierung der vierten Query